C4Hardware Documentation

Geoff Knagge
Signal Processing and Microelectronics
University of Newcastle, Australia

June 3, 2010

Contents

I Introductory Notes **5**

1 Notices **6**
1.1 Copyright . . . 6
1.2 License . . . 6
1.2.1 Preamble . . . 6
1.2.2 Terms and Conditions . . . 6
1.3 GNU General Public License version 3 . . . 7
1.3.1 Preamble . . . 7
1.3.2 Terms and Conditions . . . 8
1.4 Additional Disclaimers . . . 14

2 Introduction **15**
2.1 Typical use of the library . . . 15

3 Document Conventions **16**
3.1 Text Markup . . . 16
3.2 File names . . . 16
3.3 Markers . . . 16
3.4 Hyperlinks . . . 16

II Description of C4Hardware Components **17**

4 Overview **18**

5 Base Numeric Configuration Classes **20**
5.1 SimNumericConfiguration . . . 20
5.1.1 Static Constants . . . 20
5.1.2 Public variables . . . 21
5.1.3 Protected variables . . . 21

5.1.4 Virtual methods . . . 21
5.1.5 Public methods . . . 21
5.1.6 Protected methods . . . 21

6 Base Configuration Classes **22**
6.1 SingleValueCalculator . . . 22
6.2 DualValueCalculator . . . 22
6.3 EmulationOverrides . . . 23
6.4 DData . . . 23
6.4.1 Static Constants . . . 23
6.4.2 Public methods . . . 23
6.4.3 Public Variables . . . 24
6.4.4 Protected Variables . . . 24
6.5 SimData . . . 24

7 Base Numeric Class - SimNumericType **25**
7.1 Non-Virtual methods . . . 25
7.2 Virtual methods . . . 25
7.2.1 Duplication of Data Type . . . 25
7.2.2 Value Assignment . . . 25
7.2.3 Calculation Methods . . . 26
7.2.4 Accessor functions . . . 26
7.2.5 Miscellaneous Manipulation . . . 27
7.2.6 Static Constants . . . 27
7.2.7 Public variables . . . 27
7.2.8 Protected variables . . . 27

8 Numerical Structures **28**
8.1 SimValue . . . 28
8.1.1 Constructors . . . 28
8.1.2 Direct Call Calculation Methods . . . 28
8.1.3 Other Calculations . . . 29
8.1.4 Operators . . . 29
8.1.5 Debugging Support . . . 30
8.1.6 Miscellaneous Methods . . . 30
8.1.7 SimNumericType access . . . 30
8.1.8 Private methods . . . 30
8.2 SimValueVector . . . 30
8.2.1 Constructors . . . 31
8.2.2 Operators . . . 31
8.2.3 Direct Call Calculation Methods . . . 33
8.2.4 Vector Multiplication . . . 34
8.2.5 Debugging Support . . . 35

8.2.6 Setting Individual Cells . . . 35
8.2.7 Data Accessor Functions . . . 35
8.2.8 Miscellaneous Value assignment . . . 36
8.2.9 Vector Manipulation . . . 36
8.3 SimValueMatrix . . . 36
8.3.1 Constructors . . . 36
8.3.2 Operators . . . 37
8.3.3 Direct Call Calculation Methods . . . 38
8.3.4 Matrix Multiplication . . . 39
8.3.5 Debugging Support . . . 40
8.3.6 Data Accessor Functions . . . 41
8.3.7 Vector Manipulation . . . 41
8.3.8 Matrix Manipulation . . . 42
8.3.9 Miscellaneous Value assignment . . . 42

9 Debugging Features **43**
9.1 Labels . . . 43
9.1.1 Functionality . . . 43
9.1.2 Enabling labels . . . 44
9.2 DebugInterface . . . 44
9.3 DebugSystem . . . 44
9.3.1 Constructor . . . 45
9.3.2 Registration Tracking . . . 45
9.3.3 Main Data Log . . . 45
9.3.4 Progress Points . . . 46
9.3.5 Alternative Item Log . . . 46
9.3.6 Tracking of Calculations . . . 47

10 Exception Throwing **49**
10.1 Using Exceptions . . . 49
10.2 ExceptionSystem . . . 50
10.2.1 Static Constants . . . 50
10.2.2 Constructors . . . 51
10.2.3 Public Methods . . . 51
10.3 ExceptionPoint . . . 52
10.3.1 Constructors . . . 52
10.3.2 Public Methods . . . 52

11 Statistics Counter **53**

12 Testbench Writer **54**

III Using C4Hardware **55**

13 Getting Started **56**
13.1 The C4Hardware Automatic Configuration System . . . 56
13.1.1 Setup a debugger . . . 56
13.1.2 Setup a Statistics Module . . . 56
13.1.3 Setup a Testbench Writer . . . 57
13.2 Manually Initialising the C4Hardware Interface . . . 57
13.2.1 Setup a debugger . . . 57
13.2.2 Configure the SimNumericConfiguration . . . 58
13.2.3 Create an EmulationData . . . 58
13.2.4 Setup and configure a SimData . . . 58
13.2.5 Create a SimNumericType . . . 58
13.2.6 Other initialisation . . . 59

14 Building a Custom Numeric Type **60**
14.1 Creating a SimNumericConfiguration . . . 60
14.2 Creating a SimNumericType . . . 60
14.2.1 Constructor . . . 61
14.3 Creating a SimNumericTypeHandler . . . 62

IV Predefined Numeric Types **63**

15 SimRealFloat **64**
15.1 SimRealFloatData . . . 64
15.2 SimRealFloatConfiguration . . . 64
15.2.1 Public Methods . . . 65
15.3 SimRealFloat . . . 65

Part I

Introductory Notes

1 Notices

1.1 Copyright

This document, and the software it describes, are protected by international copyright laws and/or treaties.

The software is licensed, not sold nor given away. It may only be used for its described purpose and all other uses may constitute a breach of the license and/or copyright.

If you are aware of any breaches of copyright relating to this software, you are encouraged to contact the copyright owners via the website http://www.bit-accurate-modelling.com.

1.2 License

1.2.1 Preamble

This software is distributed under the combined terms of the GNU General Public License version 3, and also some additional terms. In the spirit of the GPL License, these additional terms do not place any restrictions on the use or distribution of the software, but do place some minor obligations on users and developers of the software.

The main themes of the additional terms are (a) that we want the project to be properly acknowledged when it is used, and (b) that we want to know about how the project is being used, so that we can determine the best ways to further evolve it for most users. This software is provided free of monetary charge as a contribution to the community, so we ask for these to be observed as a contribution back to the project.

1.2.2 Terms and Conditions

0. Definitions.

   "This License" refers to the c4hardware Additional License

   "GPL License" refers to the GNU General Public License version 3

   "Additional term" refers to a clause or subclause in this document that does not form part of the original GPL License

   "The software" refers to "the Program" and "the source code" as defined in the GNU General Public License version 3

   "we", "us", "our", "copyright holders" refers to the original authors and current project maintainers.

   All terms defined in the GPL License, that are not also defined in this license, also apply to this license.

1. Combined with GPL License

   The GNU General Public License version 3 forms part of this license

   Where the additional terms conflict with the original GPL License, the additional terms in this license shall prevail unless they invalidate the GPL License.

   If an additional term would invalidate the GPL License, then the GPL License shall prevail and the relevant additional term shall become invalid.

2. Obligations on Users

   If you use this software for a project, and it becomes and integral and useful part of that project, you should send us a short email describing your use of the software and how it is useful. This helps us understand how it is being used and how to direct future development. Other feedback, such are bugs, problems, suggestions, are welcome but completely optional.

   If the software is used in any way, whether direct or indirect, as part of any commercial application, then its use must be acknowledged in a way that would be prominant to the average user or consumer of that

application. Example of such an acknowledgments may be a message on a startup ”splash screen”, or a note in an ”about” dialog.

If the software is used in any way to produce results for a scientific publication, then an acknowledgement of the project should be made in a prominant way. An example of a suitable acknowledgement may be as a footnote, or an item in the references list of a publication.

3. Obligations on Developers

   While the GPL license allows developers to modify, add to, and redistribute the software in any way they see fit, we ask developers to contribute to the project by submitting any significant modifications or additions, for consideration in the official package.

   At our discretion, we may or may not choose to include such submissions in either the official distribution or an addition ”plug-ins” package, but in any case distribution of such changes by third parties is unrestricted.

   Any third party distributions of the official, or derived versions of, the software must contain acknowledgements and references to the original project. Where relevant, copyright notices must also be maintained.

1.3 GNU General Public License version 3

1.3.1 Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers’ and authors’ protection, the GPL clearly explains that there is no warranty for this free software. For both users’ and authors’ sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users’ freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the

special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

1.3.2 Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

   (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

   A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

   A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

   "Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

   If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction

is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant

not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

1.4 Additional Disclaimers

This statement is in addition to the disclaimers already presented in the license statements.

The software is provided solely on the grounds that it may be useful for certain requirements of an individual user, and the user accepts this upon obtaining the software. You use this software on the condition that you do so completely on an "own risk" basis, and indemnify all contributors and copyright holders against any loss, damages, or other liability arising out of its use or misuse. We do not warrant that the software is fit for any particular purpose, nor that it does not contain errors or omissions, nor that it will operate as expected. It is our intention to maintain goodwill by attempting to rectify these events should they occur, but do not warrant that we will always do so, nor that we will do so in any particular timeframe.

This document is provided as a general reference guide only and is not intended as a comprehensive training manual. As the software is continually updated, it is possible that some features described may be unavailable or do not work as described in the document. Some additional features may be available that are not described in this document. We therefore do not warrant that the software will perform as described in this document, but will endeavour to update this document as we deem appropriate.

2 Introduction

C4Hardware is a library of C++ classes that aims to provide the means to easily create software that emulates the performance of algorithms in hardware. By creating software models in a high level language, it is simpler to analyse the effects of issues such as limiting the number of bits used to represent numbers, and of optimistions that simiplify hardware at the expense of accuracy.

The focus of C4Hardware is to provide a flexible, reusable framework of classes and functions to emulate a wide range of systems. In some cases, this is done at the expense of code optimisation and, in those cases, less efficient coding methods are used in order to maximise the flexibility of the library. Contributors should check the documentation of relevant sections before attempting to optimise such code.

2.1 Typical use of the library

The intended use of C4Hardware is that it provides a middle layer of classes for use in the user's emulation project. That is, it provides an interface of classes and procedures that remain constant across all uses of the library, and should not require modification. The end user only need to write an application to call these procedures in order to implement their algorithm.

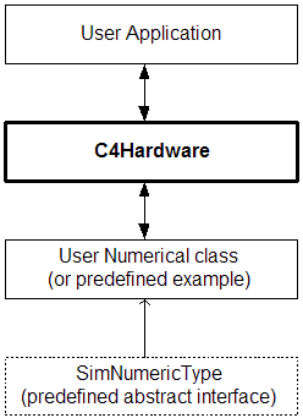

Figure 1: Layout of the components of a typical system.

At the lower level of the programming, one or more classes are needed to provide the specific numerical system required by the user. While C4Hardware also provides some examples of these, it is likely that the user will want to customise or replace these with their own versions. Each of these numerical classes will extend a predefined interface, which allows it to be used by the middle layer of classes.

3 Document Conventions

3.1 Text Markup

The following table is a guide to the color coding and font styles used throughout this document :

description	nospacing test	normal test	list item test
demo1	**example**. abcde	**example** abcde	**example** : abcde
demo2	example. abcde	example abcde	example : abcde
demo3	`example`. abcde	`example` abcde	`example` : abcde
demo4	**example**. abcde	**example** abcde	**example** : abcde
demo5	**`example`**. abcde	**`example`** abcde	
demo6	example. abcde	example abcde	example : abcde
demo7	`example`. abcde	`example` abcde	`example` : abcde
demo8	**example**. abcde	**example** abcde	**example** : abcde
demo9	“example”. abcde	“example” abcde	“example” : abcde

The correct appearance of the above examples confirms that the document formatting has compiled correctly.

3.2 File names

File names and file types can appear in a list :

- `*.abc` File type
- `c:\file.abc` File name

or as inline text : `*.abc` (File type) or `c:\file.abc` (File name)

3.3 Markers

IMPORTANT : These markers alert the reader to information that is necessary to ensure the correct operation of the software

NOTE : These markers provide useful tips and additional information

3.4 Hyperlinks

Links to other sections within the document appear as such : Document Conventions. Sometimes inter-document links may also have additional formatting applied, as described above.

Links to internet sites appear as such : Sigpromu website

Part II

Description of C4Hardware Components

4 Overview

Figure 2 shows the relationships between the various classes in the C4Hardware library. The user program only needs to directly interact with one or more of the high level data types **SimValue**, **SimValue**, and **SimValueMatrix**. To initially configure the program, an instance of **c4HardwareConfiguration** is used for transparent, parameter based, configuration. Alternatively, a "power user" may wish to create instances of the required numeric types manually and bypass the automatic configuration system.

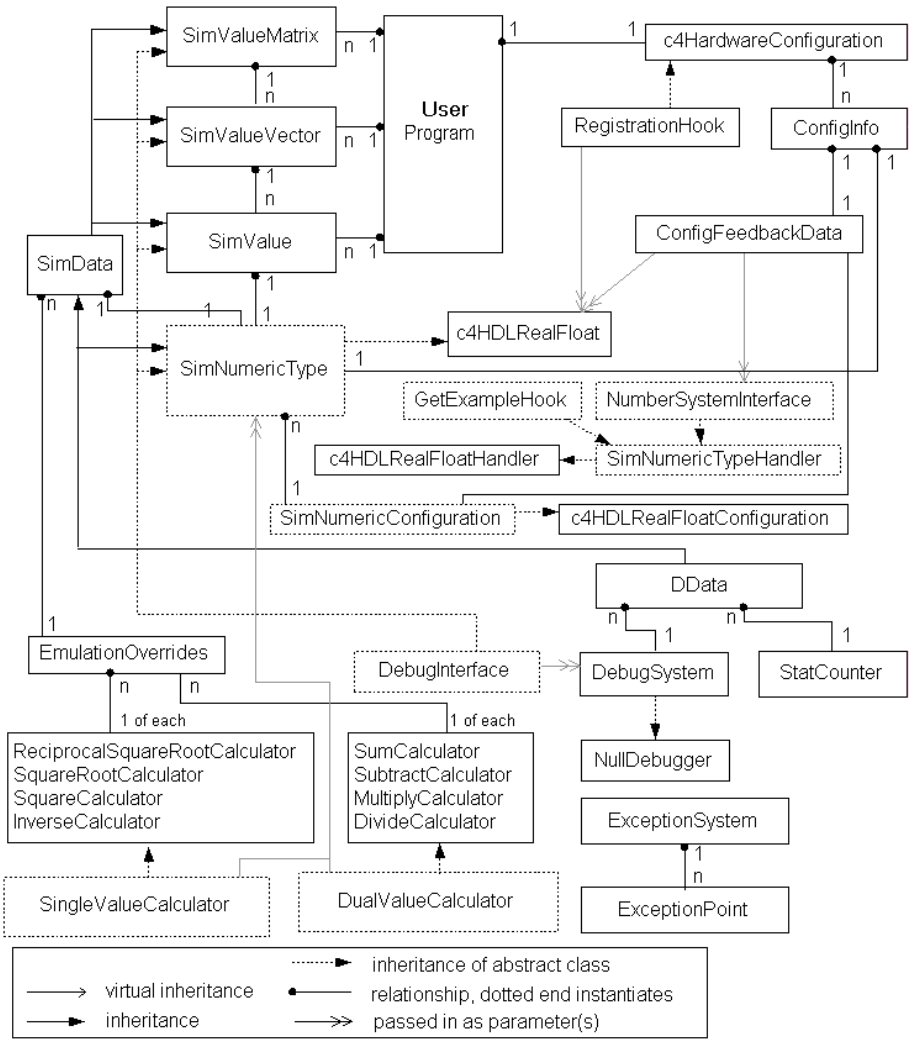

Figure 2: Relationship of classes within C4Hardware .

Each of the high level data types inherits the **SimData** class, which contains configuration information about the numeric type in use, as well as further settings inherited from **DData**. **DData** contains settings on how hardware emulation should operate, and what debugging and logging features are enabled, and is also inherited by any custom numerical classes that are created.

Whenever new high level data instances are created, the creating class typically passes itself as a parameter, possibly cast to **DData** or **SimData** , to the constructor of the new object. This allows the new object to copy configuration settings without needing specific coding by the end programmer.

SimData and **DData** are separated to avoid circular referencing in the compiler and linker. The **SimData** object contains a reference to a base **SimNumericType** so that it can be used by the high level class to create new values of the correct type. However, **SimNumericType** needs to inherit **DData** to store its configuration settings. Splitting the configuration between these two base classes avoids this issue.

One of the challenges in the design of the C4Hardware library is that the high level classes need to interact with the low level data type without knowing anything about them. This allows users of C4Hardware to create new datatypes without later having to make significant changes to the high level structures. Furthermore, those high level structures need to be able to create new data instances without knowing what that data type is.

This is achieved by defining the core data type functions in the **SimNumericType** abstract class. Part of this interface is a `duplicate` method, in which the base data type will create and return a new instance of its own type. This allows **SimValue**, **SimValue**, and **SimValueMatrix** to create and manipulate new data values without knowing anything about their class type.

Base data types are created by building classes that extend **SimNumericType**, **SimNumericTypeData**, and **SimNumericConfiguration** . For example, an included class that implements real value floating point numbers is implemented by the classes **SimRealFloat**, **SimRealFloatData**, and **SimRealFloatConfiguration** respectively. On initialisation of the main program, a "base" instance of this numeric type is created and passed into the original **SimData** instance as a **SimNumericType** . When further **SimValue**, **SimValue**, and **SimValueMatrix** instances are created with that **SimData** configuration, they will then create numeric items of the custom class type.

C4Hardware emulation occurs in one of two modes, specified by whether the `hardwareemulate` variable from **SimNumericConfiguration** is set. If it is true, then the data type should behave like it would behave in a hardware implementation, such as having restricted bit precision and overflowing numbers instead of throwing exceptions. If it is false, then the base data type should behave more like it would a traditional software application. The particulars on how this is implemented is left up to the programmer of the base data type.

When `hardwareemulate` is true, there are two further options for emulation. The software can be set to either use the emulation code defined in the base numeric type, or it is possible to override the algorithm used for specific calculations. For example, it may be desirable to replace the accurate square root with one that provides suboptimal accuracy, but performs quickly in hardware. This is done by creating a new class that implements the appropriate **SingleValueCalculator** or **DualValueCalculator** subclass (in this case **SquareRootCalculator**), and then passing it as a parameter to the appropriate method in **EmulationOverrides** (`setSqRooter`). When the square root operation is called on a **SimValue** while `hardwareemulate` is true, the new **SquareRootCalculator** object will be called instead of the square root method in the **SimNumericType** instance.

The progress of the simulated algorithm can be tracked using the **DebugSystem** class. This provides a number of logging methods, including the ability to dump data at specific points in the high level program, logging progress point markers for tracking the flow of an algorithm, and the option to log every calculation to a file.

The **StatCounter** class is currently unused, but it is intended to provide an abstract interface for the purpose of tracking statistics of an algorithm. For example, it might count number of multiplications or other operations, average number of iterations, etc.

5 Base Numeric Configuration Classes

5.1 SimNumericConfiguration

The **SimNumericConfiguration** abstract class provides an interface for passing data that specifices the behaviour of a particular numerical type. For example, it might contain information such as how many bits of precision to use, and how to handle overflow and divide-by-zero events. Normally, each instance of a numeric type sharing the same configuration will be given a reference to the same single instance of a **SimNumericConfiguration** , to reduce duplication of data.

The particular data will be specific to the **SimNumericType** that implements it, but extending this abstract class allows that data to be passed between C4Hardware objects without them needing to know for which specific numerical type it refers.

5.1.1 Static Constants

Each class extended from **SimNumericConfiguration** will need to set the `configType` variable in its constructor, so that other classes can check that it is the correct version of **SimNumericConfiguration** before forcing a cast. This is achieved by defining the `CT_` constants.

```
static const int CT_RealFloat = 1;
```

These constants allow a numerical class to be instructed on how to handle overflow anomalies :

```
// Overflow handling techniques
static const int OF_crop = 1; // discard the overflowed bits
static const int OF_staturate = 2; // set value to max/min possible
static const int OF_throw = 3; // throw an exception
```

The next set of constants were designed for the **RealFloatValue** class, but could be readily be applied to other types. They specify how the numeric class should behave when not in hardware emulation mode. It is up to the programmer of the base numeric class to determine the specific behaviour.

```
// Emulation mode constants
static const int EM_components = 0; // Use C pseudo-modelling of type
static const int EM_double = 1; // Use C internal double precision
static const int EM_single = 2; // Use C internal single precision
```

The following constants are designed to allow the specification of alternative ways of performing certain calculations. Currently unused, but provided for future expansion :

```
// Multiplication Methods
static const int MUL_standard = 1; // Use C++ * operator

// Division Methods
static const int DIV_standard = 1; // Use C++ / operator

// Square Root Methods
static const int SQRT_standard = 1; // Use C++ sqrt()

// Inversion Methods
static const int INV_standard = 1; // Use C++ 1/x

// Inverse Square Root Methods
static const int ISQRT_standard = 1;// Use C++ sqrt operator

// Squaring Methods
static const int SQR_standard = 1; // Use C++ * operator
```

5.1.2 Public variables

The following integers are for specifying the alternative ways of performing certain calculations, as indicated by the constants listed above.

```
int multiplyMethod;
int divideMethod;
int sqrtMethod;
int invertMethod;
int invsqrtMethod;
int squareMethod;
```

`bool hardwareemulate;` : determines whether or not to use hardware modelling on an attempted bit-accurate level basis, or to use a higher level C-based emulation of the numeric type. How exactly these modes are implemented depends upon the implementation of the specific **`SimNumericType`** .

`int emulationmode;` : When using `hardwareemulate=false`, this variable uses one of the `EM_` constants to determine how the **`SimNumericType`** should model the numeric type.

5.1.3 Protected variables

`int configType;` : This should be set by the constructor of the derived class, to uniquely identify that derived class type by using one of the `CT_` constants.

5.1.4 Virtual methods

`void copyConfig(const SimNumericConfiguration& src)` : This method must be implemented in derived classes, such that it copies all configuration data from the `src` object to the current instance. Typically, this method will call `copyConfigBase(src)`, and then copy any other data that is specific to the derived class.

5.1.5 Public methods

The following methods are defined in the abstract class and should not be overloaded :
`void clearNumberSystem();` : currently, this simply set `hardwareemulate` to false.
`void setEmulateMode(int md);` : sets the `emulationmode` variable to one of the defined `EM_` constants.

5.1.6 Protected methods

`copyConfigBase(const SimNumericConfiguration& src);` copies all of the **`SimNumericConfiguration`** data from the `src` object to the current instance. It is intended for use by implementations of the `copyConfig` method.

6 Base Configuration Classes

6.1 SingleValueCalculator

The **SingleValueCalculator** abstract class provides a standard interface for classes that implement customised methods of performing calculations on single values. It forms the base class for the following abstract classes

- **SquareCalculator** - calculates x^2 for a **SimNumericTypeData** x
- **SquareRootCalculator** - calculates $\sqrt{x}$ for a **SimNumericTypeData** x
- **ReciprocalSquareRootCalculator** - calculates $\frac{1}{\sqrt{x}}$ for a **SimNumericTypeData** x
- **InverseCalculator** - calculates $\frac{1}{x}$ for a **SimNumericTypeData** x

The defined virtual methods are as follows

- `virtual SimNumericTypeData calculate(const SimNumericTypeData& inval, const SimNumericConfiguration& outconfig)=0;` - Operates on the provided input `inval`, and uses `outconfig` to generate a new SimNumericTypeData as the output.
- `virtual void calculate(SimNumericTypeData& inval)=0;` - uses the input as the output
- `virtual void calculate(const SimNumericTypeData& inval, SimNumericTypeData& outval)=0;` - operates on the `inval` input and writes to `outval`.
- `virtual void calculateArray(const SimNumericTypeData** inval, SimNumericTypeData** outval)=0;` - operates on an array of elements, writing the result of each `inval` element to a corresponding element in `outval`.

6.2 DualValueCalculator

The **DualValueCalculator** abstract class provides a standard interface for classes that implement customised methods of performing calculations with two operands. It forms the base class for the following abstract classes

- **SumCalculator** - calculates $a + b$ for **SimNumericTypeData** a and b
- **MultiplyCalculator** - calculates $a * b$ for **SimNumericTypeData** a and b
- **DivideCalculator** - calculates a/b for **SimNumericTypeData** a and b
- **SubractCalculator** - calculates $a - b$ for **SimNumericTypeData** a and b

- `virtual SimNumericTypeData calculate(const SimNumericTypeData& rightval, const SimNumericTypeData& leftval, const SimNumericConfiguration& outconfig)=0;` - Operates on the provided inputs `leftval` and `rightval`, and uses `outconfig` to generate a new SimNumericTypeData as the output.
- `virtual void calculate(const SimNumericTypeData& rightval, const SimNumericTypeData& leftval,SimNumericTy outval)=0;` - Operates on the provided inputs `leftval` and `rightval`, and writes to `outval`
- `virtual void calculateArray(const SimNumericTypeData** rightval, const SimNumericTypeData** lefttval, SimNumericTypeData** outval)=0;` - operates on an array of elements, writing the result of each pair of `leftval` and `rightval` elements to a corresponding element in `outval`.

6.3 EmulationOverrides

The **EmulationOverrides** class defines the following public variables to specify **SingleValueCalculator** and **DualValueCalculator** classes that are to be used to override various types of calculations :

```
SquareRootCalculator* sqrtcalc;
SquareCalculator* sqrcalc;
ReciprocalSquareRootCalculator* invsqrtcalc;
InverseCalculator* invcalc;
SubtractCalculator* subtractcalc;
SumCalculator* sumcalc;
MultiplyCalculator* multcalc;
DivideCalculator* divcalc;
```

Although the variables are public, it is recommended to use the following methods to modify them. This will ensure that any future configuration that is added to the classes will be applied.

```
void setInverter(InverseCalculator* m);
void setRecipSqRooter(ReciprocalSquareRootCalculator* m);
void setSqRooter(SquareRootCalculator* m);
void setSquarer(SquareCalculator* m);
void setMultiplier(MultiplyCalculator* m);
void setDivider(DivideCalculator* m);
void setSummer(SumCalculator* m);
void setSubtractor(SubtractCalculator* m);
```

Override classes are used by **SimValue** classes when the `hardwareemulate` variable from **SimNumericConfiguration** is set to true and the relevant pointer is set to a non-NULL value.

6.4 DData

The **DData** base class provides general information about the emulation settings in the project

6.4.1 Static Constants

```
static const int OB_SimValue = 1;
static const int OB_FloatValue = 1;
static const int OB_RealVector = 2;
static const int OB_RealMatrix = 3;
```

6.4.2 Public methods

- `void copyDDataConfig(const DData &src);` - copies most of the data from the `src` to the current instance. Does not copy the `debugger` or `label` variables.
- `void setIndexing(bool indexingStartsAtOne);` - sets the indexing mode. If true, then the first cell in a vector, or row and column in a matrix, is indexed by 1. If false, those items are indexed by 0.

The following methods are described in Section 9.3 on the **DebugSystem** class.

```
virtual void setLabel(const char* txt);
virtual void afterSetLabel();
void giveTempLabel();
void setLabel(const char* txt, const char* txt2);
void copyLabel(char* txt);
void copyLabelAsCopy(char* txt);
char* getID() const;
```

```
void copyLabelTo(char* txt) const;
```

6.4.3 Public Variables

- `DebugSystem* debugger;` - a reference to the **`DebugSystem`** to be used by this object
- `long debugID;` - is a unique identifier assiged by the debugger, and used to identify the object instance in debugging logs.

6.4.4 Protected Variables

- `EmulationOverrides* emulation;`
- `StatCounter* stats;`
- `bool countStats;`
- `int obtype;` - should be set to one of the `_OB` constants by classes that extend **`DData`**
- `bool indexFromOne;` - if true, then the first cell in a vector, or row and column in a matrix, is indexed by 1. If false, those items are indexed by 0.
- `int indexoffset;` - numerically indicated the index that refers to the first cell, row, or column in a vector or matrix

The following variables are used for implementing the debugging system :

```
char* label;
bool labelCopied;
```

The `label` is a human readable string to provide a descriptive name for printing to the debugging logs. The `labelCopied` variable indicates whether this instance created its own label (in which case it must destroy it) or has copied a reference to a label from another object (in which case it must NOT delete or modify it)

6.5 SimData

`SimData` extends the **`DData`** class

- `void copySimConfig(const SimData &src);` - calls `copyDDataConfig(src)` and also copies the `baseconfig` pointer from the `src` object.
- `void setBaseConfig(SimNumericType* bc);` - sets the `baseconfig` variable

One protected variable is also defined :

- `SimNumericType* baseconfig;` - references the **`SimNumericType`** to be used for creating new data items.

7 Base Numeric Class - SimNumericType

SimNumericType is the main class upon which all numeric types are based, and is the interface between the middle later C4Hardware classes, and the lower level customised classes. It brings together all of the customised and extended base configuration classes and defined the behaviour of the main types of calculations for this numeric type.

Each **SimValue** instance references one **SimNumericType** , and calls on its virtually defined methods to perform mathematical operations. These virtual methods are redefined in the custom numeric classes and are invoked by these calls.

The class delaration is as follows : `class SimNumericType : public DData, public virtual SimNumericTypeData, public DebugInterface`

It is important to note the virtual inheritance of **SimNumericTypeData** . This is because classes the extend **SimNumericType** also need to extend their own version of **SimNumericTypeData** , creating two inheritance paths. By declaring both as virtual, there will be no ambiguity as both paths will refer to the same data.

7.1 Non-Virtual methods

The `bool hardwareEmulate();` routine simply returns the state of the hardwareemulate variable. It should not be overloaded.

7.2 Virtual methods

All of the methods in this section must be defined in any class that inherits from **SimNumericType** , even if they aren't used or are irrelevant to the data type. For example, `imagValue()` needs to be implemented, even though it will always return zero for a real-only numeric type.

7.2.1 Duplication of Data Type

The following are perhaps the two simplest, but most powerful methods in the C4Hardware library.

```
 virtual SimNumericType* duplicate() =0;
virtual SimNumericType* duplicate(char* txt, bool labelAsCopy=false) =0;
```

These routines allow higher level classes, such as **SimValue** , **SimValueVector** , and **SimValueMatrix** to create new instances of this data type without knowing anything about it. Therefore, when implemented, these methods must do all of the configuration necessary to produce and return a reference to an intialised, usable, and independant, version of itself.

7.2.2 Value Assignment

Each of the following methods sets the numerical value of the current instance. They should not affect any non-related data.

```
 virtual void assign(long a)=0;
virtual void assign(double a)=0;
virtual void assign(int a)=0;
virtual void assign(const SimNumericType &a)=0;
virtual void assign(const SimNumericType* a)=0;
```

7.2.3 Calculation Methods

The abstract class defines a series of methods for the most common calculations, as follows :

```
virtual void doSubtract(const SimNumericType& a)=0;
virtual void doAdd(const SimNumericType& aa)=0;
virtual void doMultBy(const SimNumericType& a)=0;
virtual void doMultWith(const SimNumericType& a)=0;
virtual void doDivide(const SimNumericType& a)=0;

virtual void doSubtract_always(const SimNumericType& a)=0;
virtual void doAdd_always(const SimNumericType& aa)=0;
virtual void doMultBy_always(const SimNumericType& a)=0;
virtual void doMultWith_always(const SimNumericType& a)=0;
virtual void doDivide_always(const SimNumericType& a)=0;

virtual void setWithReciprocalSqrt(const SimNumericType &a)=0;
virtual void setWithSquare(const SimNumericType &a)=0;
virtual void setWithInverse(const SimNumericType &a)=0;
virtual void setWithSqrt(const SimNumericType &a)=0;

virtual void setWithReciprocalSqrt_always(const SimNumericType &a)=0;
virtual void setWithSquare_always(const SimNumericType &a)=0;
virtual void setWithInverse_always(const SimNumericType &a)=0;
virtual void setWithSqrt_always(const SimNumericType &a)=0;

virtual void setWithReciprocalSqrt_Hardware(const SimNumericType &a)=0;
virtual void setWithSquare_Hardware(const SimNumericType &a)=0;
virtual void setWithInverse_Hardware(const SimNumericType &a)=0;
virtual void setWithSqrt_Hardware(const SimNumericType &a)=0;
```

The implementation of each routine must store the result of each operation in the current instance. If two operand operation, the parameter is to be used as the second operand (i.e. the quantity being taken away in subtract). The only exception is for `doMultWith`, where the parameter should by used as the multiplicand.

Each operation has up to three sets of methods to implement :

- Methods with the `_always` suffix, as the name suggests, are always called when a particular operation is triggered. Note that this may happen in addition to one of the other methods.
- Methods with the `_Hardware` suffix are called when the `hardwareemulate` variable from **`SimNumericConfiguration`** is set to true. However, they are NOT called if there is an override set in the **`EmulationOverrides`** class.
- Methods with no suffix are called when there is no `_Hardware` version, or if the `hardwareemulate` variable from **`SimNumericConfiguration`** is set to false. However, they are NOT called if there is an override set in the **`EmulationOverrides`** class.

In summary, either the `_Hardware` **or** no-suffix version may be called if there is no override, and the `_always` version is always called regardless of overrides.

7.2.4 Accessor functions

In each of the following, it will be up to the programmer of the class to determine how exactly each of these methods should behave. However, the following are a guideline of what is intended :

- `virtual bool almostzero() const=0;` : should return true if the value of the quantity is "close to" zero.

- `virtual double value() const=0;` : returns a double representing the value of the magnitude and sign of the quantity
- `virtual double realValue() const=0;` : returns the magnitude and sign of the real part of the quantity
- `virtual double imagValue() const=0;` : returns the magnitude and sign of the imaginary part of the quantity
- `virtual int getExponent() const=0;` : returns the exponent of a floating point number, or an equivalent
- `virtual bool isNegative() const=0;` : returns true if the value is negative

In all of the above, the values returned may depend on the value assigned to `emulationmode` in **SimNumericConfiguration** . However, for the following, the implementation should assume that `emulationmode` is set to zero.

- `virtual double valueOfComponents()=0;`

As an example of the use of this, the `_always` routines in the calculation methods might keep a double or single precision value calculated in pure C/C++, while the other routines emulate a hardware model. If emulation mode is set to `SimNumericConfiguration::EM_double`, then the `value()` and `valueOfComponents()` could be compared to evaluate the differences in the two methods of calculation.

7.2.5 Miscellaneous Manipulation

- `virtual void adjustExponent(int adj)=0;` : adjust the quantity by 2^{adj}.
- `virtual void generateRandom()=0;` : create a random quantity.
- `virtual void dump() const=0;` : logs the current value to the debugger, if it has been defined.
- `virtual void negate()=0;` : takes the negative of the current value.

7.2.6 Static Constants

Each class extended from the **SimNumericTypeData** will need to set the `numericType` variable in its constructor, so that other classes can check it is the correct version of **SimNumericTypeData** before forcing a cast.

```
static const int NT_RealFloat = 1;
```

7.2.7 Public variables

`int numericType;` : This should be set by the constructor of the derived class, to uniquely identify that derived class type by using one of the `NT_` constants.

7.2.8 Protected variables

`SimNumericConfiguration* numeric;` : provides a pointer to a **SimNumericConfiguration** object for this class. This variable must be validly set by the constructor of the derive classes.

8 Numerical Structures

8.1 SimValue

The **SimValue** class represents the most basic numerical quantity in a project, and extends the **SimData** and **DebugInterface** classes.

8.1.1 Constructors

```
SimValue(const SimData &config, const char* txt=NULL);
SimValue(const SimData *config, const char* txt=NULL);
SimValue(const SimValue* src, const char* txt=NULL);
SimValue(const SimValue& src, const char* txt=NULL, bool allowAutoDuplicate=true);
```

The constructor must be given a base configuration to copy to the new instance, which can be in the form of a **SimData** , or another **SimValue** . In all cases, an optional label may be supplied as a second parameter.

If a **SimValue** is supplied, there exists the option to automatically create a duplicate of the label. In this case, the new instance will be labelled the same as the source, but with the word “duplicate” appended.

8.1.2 Direct Call Calculation Methods

These methods provide the same functionality as the overloaded operators, but may provide greater efficiency of compiled code, and are more flexible to use. Each method sets the called instance with the result of operating on the supplied parameters. The first parameter represents the left operand in an expresion, and the second parameter is the right operand.

```
void setWithDivide(const SimValue *a, const SimValue*b);
void setWithDivide(const SimValue *a, int b);
void setWithDivide(const SimValue *a, long b);
void setWithDivide(const SimValue *a, double b);
void setWithDivide(int b, const SimValue *a);
void setWithDivide(long b, const SimValue *a);
void setWithDivide(double b, const SimValue *a);
void setWithDivide(const SimValue &a, const SimValue*b);
void setWithDivide(const SimValue *a, const SimValue&b);
void setWithDivide(const SimValue &a, const SimValue&b);
void setWithDivide(const SimValue &a, int b);
void setWithDivide(const SimValue &a, long b);
void setWithDivide(const SimValue &a, double b);
void setWithDivide(int b, const SimValue &a);
void setWithDivide(long b, const SimValue &a);
void setWithDivide(double b, const SimValue &a);

void setWithMultiply(const SimValue *a, const SimValue*b);
void setWithMultiply(const SimValue *a, int b);
void setWithMultiply(const SimValue *a, long b);
void setWithMultiply(const SimValue *a, double b);
void setWithMultiply(int a, const SimValue *b);
void setWithMultiply(long a, const SimValue *b);
void setWithMultiply(double a, const SimValue *b);
void setWithMultiply(const SimValue &a, const const SimValue&b);
void setWithMultiply(const SimValue &a, const SimValue*b);
void setWithMultiply(const SimValue *a, const SimValue&b);
void setWithMultiply(const SimValue &a, int b);
void setWithMultiply(const SimValue &a, long b);
```

```
void setWithMultiply(const SimValue &a, double b);
void setWithMultiply(int a, const SimValue &b);
void setWithMultiply(long a, const SimValue &b);
void setWithMultiply(double a, const SimValue &b);

void setWithAdd(const SimValue *a, const SimValue*b);
void setWithAdd(const SimValue *a, int b);
void setWithAdd(const SimValue *a, long b);
void setWithAdd(const SimValue *a, double b);
void setWithAdd(const SimValue &a, const SimValue*b);
void setWithAdd(const SimValue *a, const SimValue&b);
void setWithAdd(const SimValue &a, const SimValue&b);
void setWithAdd(const SimValue &a, int b);
void setWithAdd(const SimValue &a, long b);
void setWithAdd(const SimValue &a, double b);
```

8.1.3 Other Calculations

Provision is also made for some of the less common calculations. The following routines set the called instance with the result obtained by operating on the provided parameter.

```
void setWithSqrt(const SimValue &a);
void setWithSquare(const SimValue &a);
void setWithInverse(const SimValue &a);
void setWithReciprocalSqrt(const SimValue &a);
```

These routines provide the same functionality, but instead operate on the called instance and return a new **SimValue** as the result:

```
SimValue getSqrt() const;
SimValue getSquare() const;
SimValue getInverse() const;
SimValue getReciprocalSqrt() const;
```

8.1.4 Operators

SimValue implements the most common mathematical operators. There are two types of operators, being the ones that perform an operation between two different operand and assign to a potentially different destination (such as $a = b + c$), and the ones that use the destination as one of the operands (such as $a+ = b$).

The $a = b + c$ type operators pass their result by value, rather then by reference. In the case of **SimValue** objects, this means creating a temporary variable to hold the result, and then the compiler creates a copy of that result before calling an assign operator to apply it to the destination variable. This is rather inefficient for code speed, but allows for simple programming especially in compound expressions).

Each operator is implemented with five overloads, to accept `const SimValue &a`, `SimValue* a`, `int a`, `long a`, or `double a` as the parameter. The operator is called on the first operand in the expression, for example the `a` object in `c=a+b`.

```
bool operator<(const SimValue &a);
SimValue operator/(const SimValue &a);
SimValue operator-(const SimValue &a);
SimValue operator+(const SimValue &a);
SimValue operator*(const SimValue &a);
```

The equality type operators are called on the destination operand, which is also the first operand in the expression. Since these operators pass by reference, they are just as efficient as using standard methods. These are also overloaded to accept the same five types of operands.

```
SimValue &operator/=(const SimValue &a);
SimValue &operator*=(const SimValue &a);
```

```
SimValue &operator+=(const SimValue &a);
SimValue &operator-=(const SimValue &a);
SimValue &operator=(const SimValue &a);
```

8.1.5 Debugging Support

The **SimValue** class implements the **DebugInterface** and provides debugging support with the following :

- `void logItem(const char* desc=NULL, const char* indent=NULL);` - calls logItem of the **DebugSystem** instance, if defined
- `void dump() const;` - writes the value of the current instance to the main logs
- `void afterSetLabel();` - passes the new label on to the SimNumericType for this instance.
- `void getCalcLabel(char* txt) const;` - returns the label of this instance
- `void getCalcLogValue(char* txt) const;` - returns a string for the calculation tracking and item logs, representing the value of the current instance.

8.1.6 Miscellaneous Methods

- `SimNumericType* getItem() const;` - returns a reference to the **SimNumericType** associated with this instance.
- `DebugSystem* getDebugger() const;` - returns a reference to the **DebugSystem** being used.

8.1.7 SimNumericType access

The following methods provide access to the **SimNumericType** instance owned by the current **SimValue** (via **SimData**). They simply call the same method in that instance, returning the results if any.

```
 bool almostzero() const
double value() const
double valueOfComponents() const
int getExponent() const;
bool isNegative() const;
void generateRandom();
void adjustExponent(int adj);
void negate();
```

8.1.8 Private methods

These are the methods that do the actual calculation and logging work in the **SimValue** class. All of the other operators and calculation methods call these to obtain their results.

```
void doAdd(const SimNumericType& a);
void doSubtract(const SimNumericType& a);
void doMultBy(const SimNumericType& a);
void doMultWith(const SimNumericType& a);
void doDivide(const SimNumericType& a);
```

8.2 SimValueVector

The **SimValueVector** class represents mathematical vectors, that are made up of cells containing instances of **SimValue** . It extends the **SimData** and **DebugInterface** classes.

8.2.1 Constructors

```
 SimValueVector(const SimData* config, int cellcnt, bool col, bool setgrowable=true, int maxsize=0);
SimValueVector(const SimData& config, int cellcnt, bool col, bool setgrowable=true, int maxsize=0);
SimValueVector(const SimData* config, double* initvals, int cellcnt, bool col, bool setgrowable=true,
int maxsize=0);
SimValueVector(const SimData& config, double* initvals, int cellcnt, bool col, bool setgrowable=true,
int maxsize=0);
SimValueVector(int cellcnt, SimData* config, bool col, const char* txt, const char* activelabel=NULL,
bool setgrowable=true, int maxsize=0);
SimValueVector(int cellcnt, const SimData& config, bool col, const char* txt, const char* activelabel=NULL,
bool setgrowable=true, int maxsize=0);
SimValueVector(int cellcnt, SimData* config, double* initvals, bool col, const char* txt, const
char* activelabel=NULL, bool setgrowable=true, int maxsize=0);
SimValueVector(int cellcnt, const SimData& config, double* initvals, bool col, const char* txt,
const char* activelabel=NULL, bool setgrowable=true, int maxsize=0);
SimValueVector(const SimValueVector &src, const char* txt=NULL, const char* activelabel=NULL);
SimValueVector(const SimValueVector* src, const char* txt=NULL, const char* activelabel=NULL);
```

The constructor must be given a base configuration to copy to the new instance, which can be in the form of a **SimData** , or another **SimValueVector** . If a **SimValueVector** is supplied, it is used to provide the initial configuration parameters.

Each new **SimValueVector** needs to have specified the initial number of cells (`cellcnt`), and whether it is a column vector (`col=true`) or a row vector (`col=false`).

An optional parameter is to specify whether a vector is "growable". If `setgrowable` is true (the default), then the vector may be resized to any required number of cells. If it is set to false, then the `maxsize` parameter specifies the maximum number of cells that are permitted in this vector.

Initial values may be provided, by passing a reference to an array of type `double` to the `initvals` parameter. Care should be taken that this array is at least as long as the number of initial cells specified.

Labels may be assigned via the `txt` and `activelabel` parameters. The `txt` string is used to set the label of the new **SimValueVector** . If provided, `activelabel` should be a `printf` style string that contains one `%d` item. This `activelabel` is then used to label all of the cells within the vector. For example, an `activelabel` of "fvec, cell %d" would result in cells labelled "fvec, cell 1", "fvec, cell 2", etc. If no `activelabel` is supplied, then the cell are labelled with the vector label and thier index in parentheses (e.g. "fvec(1)").

8.2.2 Operators

As with **SimValue** , **SimValueVector** implements the basic mathematical operators. There are two types of operators, being the ones that perform an operation between two different operands and assign to a potentially different destination (such as $a = b + c$), and the ones that use the destination as one of the operands (such as $a+ = b$).

The $a = b+c$ type operators pass their result by value, rather then by reference. In the case of **SimValueVector** objects, this means creating a temporary variable to hold the result, and then the compiler creates a copy of that result before calling an assign operator to apply it to the destination variable. This is rather inefficient for code speed, but allows for simple programming especially in compound expressions.

```
SimValueVector operator +(const SimValueVector &a); SimValueVector operator +(const SimValueVector
*a);
SimValueVector operator +(const SimValue &a);
SimValueVector operator +(const SimValue *a);
SimValueVector operator +(double a);
SimValueVector operator +(long a);
SimValueVector operator +(int a);
```

```
SimValueVector operator -(const SimValueVector &a);
SimValueVector operator -(const SimValueVector *a);
SimValueVector operator -(const SimValue &a);
SimValueVector operator -(const SimValue *a);
SimValueVector operator -(double a);
SimValueVector operator -(long a);
SimValueVector operator -(int a);

SimValueVector operator *(const SimValueVector &a);
SimValueVector operator *(const SimValueVector *a);
SimValueVector operator *(const SimValue &a);
SimValueVector operator *(const SimValue *a);
SimValueVector operator *(double a);
SimValueVector operator *(long a);
SimValueVector operator *(int a);

SimValueVector operator /(const SimValue &a);
SimValueVector operator /(const SimValue *a);
SimValueVector operator /(double a);
SimValueVector operator /(long a);
SimValueVector operator /(int a);
```

NOTE : When another **SimValueVector** is supplied to the multiplication operator, it works as a MATLAB "dot multiply" function, multiplying corresponding elements of each element. To obtain a true vector multiplication, consider the `vectorMultiply` methods.

The equality type operators are called on the destination operand, which is also the first operand in the expression. Since these operators pass by reference, they are just as efficient as using standard methods.

```
SimValueVector &operator*=(const SimValueVector &a);
SimValueVector &operator*=(const SimValueVector *a);
SimValueVector &operator*=(const SimValue &a);
SimValueVector &operator*=(const SimValue *a);
SimValueVector &operator*=(double a);
SimValueVector &operator*=(long a);
SimValueVector &operator*=(int a);

SimValueVector &operator/=(const SimValueVector &a);
SimValueVector &operator/=(const SimValueVector *a);
SimValueVector &operator/=(const SimValue &a);
SimValueVector &operator/=(const SimValue *a);
SimValueVector &operator/=(double a);
SimValueVector &operator/=(long a);
SimValueVector &operator/=(int a);

SimValueVector &operator+=(const SimValueVector &a);
SimValueVector &operator+=(const SimValueVector *a);
SimValueVector &operator+=(const SimValue &a);
SimValueVector &operator+=(const SimValue *a);
SimValueVector &operator+=(double a);
SimValueVector &operator+=(long a);
SimValueVector &operator+=(int a);

SimValueVector &operator-=(const SimValueVector &a);
SimValueVector &operator-=(const SimValueVector *a);
SimValueVector &operator-=(const SimValue &a);
SimValueVector &operator-=(const SimValue *a);
SimValueVector &operator-=(double a);
```

```
SimValueVector &operator-=(long a);
SimValueVector &operator-=(int a);

SimValueVector &operator=(const SimValueVector &a);
SimValueVector &operator=(const SimValueVector *a);
```

NOTE : The following operators will resize the vector to length 1 and any old copies of the data reference will become invalid

```
SimValueVector &operator=(const SimValue &a);
SimValueVector &operator=(const SimValue *a);
SimValueVector &operator=(double a);
SimValueVector &operator=(long a);
SimValueVector &operator=(int a);
```

8.2.3 Direct Call Calculation Methods

These methods provide the same functionality as the overloaded operators, but may provide greater efficiency of compiled code, and are more flexible to use. Each method sets the called instance with the result of operating on the supplied parameters. The first parameter represents the left operand in an expresion, and the second parameter is the right operand. The "DotMultiply" routines are equivalent to the MATLAB `.*` statement, where corresponding pairs of elements are multiplied instead of performing a full vector multiplication.

NOTE : To multiply a vector and a matrix together, refer to the **`SimValueMatrix`** class, even if the result is known to be a vector. This needs to be done to avoid circular referencing of classes.

```
void setWithDotMultiply(const SimValueVector *a, const SimValueVector *b);
void setWithDotMultiply(const SimValueVector *a, const SimValueVector &b);
void setWithDotMultiply(const SimValueVector &a, const SimValueVector &b);
void setWithDotMultiply(const SimValueVector &a, const SimValueVector *b);

void setWithMultiply(const SimValueVector *a, const SimValue *b);
void setWithMultiply(const SimValueVector *a, const SimValue &b);
void setWithMultiply(const SimValueVector *a, long b);
void setWithMultiply(const SimValueVector *a, int b);
void setWithMultiply(const SimValueVector *a, double b);
void setWithMultiply(const SimValue *a, const SimValueVector *b);
void setWithMultiply(const SimValue &a, const SimValueVector *b);
void setWithMultiply(double a, const SimValueVector *b);
void setWithMultiply(int a, const SimValueVector *b);
void setWithMultiply(long a, const SimValueVector *b);


void setWithDivide(const SimValueVector *a, const SimValue *b);
void setWithDivide(const SimValueVector *a, const SimValue &b);
void setWithDivide(const SimValueVector *a, long b);
void setWithDivide(const SimValueVector *a, int b);
void setWithDivide(const SimValueVector *a, double b);

void setWithAdd(const SimValueVector *a, const SimValueVector *b);
void setWithAdd(const SimValueVector *a, const SimValueVector &b);
void setWithAdd(const SimValueVector *a, const SimValue *b);
void setWithAdd(const SimValueVector *a, const SimValue &b);
void setWithAdd(const SimValueVector *a, long b);
void setWithAdd(const SimValueVector *a, int b);
void setWithAdd(const SimValueVector *a, double b);

void setWithSubtract(const SimValueVector *a, const SimValueVector *b);
```

```
void setWithSubtract(const SimValueVector *a, const SimValueVector &b);
void setWithSubtract(const SimValueVector *a, const SimValue *b);
void setWithSubtract(const SimValueVector *a, const SimValue &b);
void setWithSubtract(const SimValueVector *a, long b);
void setWithSubtract(const SimValueVector *a, int b);
void setWithSubtract(const SimValueVector *a, double b);
```

8.2.4 Vector Multiplication

All of the following declarations are declared as `static void` and the end of each declaration is `, bool oneIsTransposed=false, bool doNeg=false);`. These details are omitted for clarity. For example, the full first declaration is

```
static void vectorMultiply(int cellsToUse, SimValue &target, const SimValueVector &a, const SimValueVector
&b, bool oneIsTransposed=false, bool doNeg=false);
```

```
vectorMultiply(int cellsToUse, SimValue &target, const SimValueVector &a, const SimValueVector
&b
vectorMultiply(int cellsToUse, SimValue &target, const SimValueVector *a, const SimValueVector
&b
vectorMultiply(int cellsToUse, SimValue &target, const SimValueVector &a, const SimValueVector
*b
vectorMultiply(int cellsToUse, SimValue &target, const SimValueVector *a, const SimValueVector
*b

vectorMultiply(int cellsToUse, SimValue *target, const SimValueVector &a, const SimValueVector
&b
vectorMultiply(int cellsToUse, SimValue *target, const SimValueVector *a, const SimValueVector
&b
vectorMultiply(int cellsToUse, SimValue *target, const SimValueVector &a, const SimValueVector
*b
vectorMultiply(int cellsToUse, SimValue *target, const SimValueVector *a, const SimValueVector
*b

vectorMultiply(SimValue &target, const SimValueVector &a, const SimValueVector &b
vectorMultiply(SimValue &target, const SimValueVector *a, const SimValueVector &b
vectorMultiply(SimValue &target, const SimValueVector &a, const SimValueVector *b
vectorMultiply(SimValue &target, const SimValueVector *a, const SimValueVector *b

vectorMultiply(SimValue *target, const SimValueVector &a, const SimValueVector &b
vectorMultiply(SimValue *target, const SimValueVector *a, const SimValueVector &b
vectorMultiply(SimValue *target, const SimValueVector &a, const SimValueVector *b
vectorMultiply(SimValue *target, const SimValueVector *a, const SimValueVector *b
```

The following are also declared as `static`, with the same ending tothe declaration as above.

```
SimValue vectorMultiply(const SimValueVector &a, const SimValueVector &b
SimValue vectorMultiply(const SimValueVector *a, const SimValueVector &b
SimValue vectorMultiply(const SimValueVector &a, const SimValueVector *b
SimValue vectorMultiply(const SimValueVector *a, const SimValueVector *b

SimValue vectorMultiply(int cellsToUse, const SimValueVector &a, const SimValueVector &b
SimValue vectorMultiply(int cellsToUse, const SimValueVector *a, const SimValueVector &b
SimValue vectorMultiply(int cellsToUse, const SimValueVector &a, const SimValueVector *b
SimValue vectorMultiply(int cellsToUse, const SimValueVector *a, const SimValueVector *b
```

In each of the above, `a` and `b` are the vectors to be multiplied. If `oneIsTransposed` is false, then ane should be a column vector and one should be a row vector. If `oneIsTransposed` is true, then both must be row vectors

or both must be column vectors. the `doNeg` flag indicates whether to negate the final result. Where used, the `cellsToUse` input specifies how many cells should be used from each matrix, starting from the first cell.

8.2.5 Debugging Support

The **`SimValueVector`** class implements the **`DebugInterface`** and provides debugging support with the following :

- `void logItem(const char* desc=NULL, bool indent=false) const;` - calls logItem of the **`DebugSystem`** instance, if defined
- `void dump() const;` - writes the value of the current instance to the main logs
- `void dump(bool transmarker) const;` - as above, but the parameter specified whether to write the transpose marker (′) when dumping column vectors (all vectors are printed to logs as rows)
- `void setupLabel(const char* txt, const char* activetxt);` - Allows for the initialisation or changing of the label.
- `void getCalcLabel(char* txt) const;` - returns the label of this instance
- `void getCalcLogValue(char* txt) const;` - returns a string for the calculation tracking and item logs, representing the value of the current instance.

8.2.6 Setting Individual Cells

Each of the following allow the setting of the contents of individual cells in a vector :

```
void setCell(int row, const SimValue* dat, bool copyLabel = false);
void setCell(int row, const SimValue& dat, bool copyLabel = false);
void setCell(int row, double dat);
void setCellIndex0(int row, const SimValue* dat, bool copyLabel = false);
void setCellIndex0(int row, const SimValue& dat, bool copyLabel = false);
void setCellIndex0(int row, double dat);
```

In each case, `row` indexes the required cell. For the `setCellIndex0` methods, this is always zero-based, with index 0 referring to the first cell. For the `setCell` routines, it is only zero-based if the `indexFromOne` variable inherited from **`DData`** is false. If it is true (a `setIndexing(true)` call was made), then indexing is one-based, with index 1 referring to the first cell.

8.2.7 Data Accessor Functions

- `void fillArray(double* d); const` - fills the array referenced by `d` with doubles representing the values of the vector. Care must be taken to ensure the the array is large enough.
- `SimValue* getCell(int roworcol);` - Returns a reference to the cell indexed by `roworcol`. Indexing is zero or one based, depending on the setting of the `indexFromOne` variable inherited from **`DData`**
- `SimValue* getCellIndex0(int roworcol) const;` - Returns a reference to the cell indexed by `roworcol`. Indexing is zero based.
- `SimValue** getData();` - Returns a reference to the array of vector cells
- `SimValue operator[](int i) const;` - same as `getCell(int roworcol);`
  NOTE : Be careful using the indexing operator. If the variable is a pointer to the **`SimValueVector`** then make sure that the code is accessing the operator, not indexing an array pointed to by the variable. e.g. if you declared `SimValueVector* f;` then you would need to use `(*f)[1].dump();` or `f[0][1].dump();`.
- `SimValue* valueof();` - returns the value of the first cell in the vector.

- `bool isColumn() const;` - returns true if it is a column vector.
- `int getsize() const;` - returns the current number of cells in the vector.

8.2.8 Miscellaneous Value assignment

- `void negate();` - negates all of the cells in the vector
- `void ones();` - sets all of the cells to 1
- `void zeros();` - sets all of the cells to 0
- `void zeros(int cnt);` - resizes the vector to `cnt`, and sets all of the cells to 0
- `void zerosAndAOne(int onepos);` - sets all of the cells to 0, except for cell `onepose`, which is set to 1
- `void generateRandom();` - sets all of the cells to random values

8.2.9 Vector Manipulation

- `void setWithVector(const SimValueVector &src, bool negate );` - copies the data from `src` into the current vector, optionally negating the values.
- `void setWithVector(const SimValueVector *src, bool negate );` - as above
- `void deleteCell(int src);` - deletes the cell indexed by `src`. Indexing is zero or one based, depending on the setting of the `indexFromOne` variable inherited from **`DData`**
- `void resize(int newcells);` - resizes the vector, adding or deleting cells as needed
- `void resize(int newcells, const SimData &config);` - resizes the vector, using `config` as the base configuration for the new cells
- `void setOrientation(bool isColumn);` - changes whether the vector is a row vector or a column vector

8.3 SimValueMatrix

The **`SimValueMatrix`** class represents mathematical matrices, that are made up of an array of **`SimValueVector`** . The vectors may be stored as column vectors or row vectors, depending on the configuration chosen at initialisation. The class extends the **`SimData`** and **`DebugInterface`** classes.

8.3.1 Constructors

Each of the following declarations ends with

```
, int height, int width, bool asColumns, bool setgrowable=true, int setmaxrows=0, int setmaxcols=0);,
```

which has been omitted for clarity.

```
SimValueMatrix(const SimData& config
SimValueMatrix(const SimData* config
SimValueMatrix(const SimData& config, double* initdata
SimValueMatrix(const SimData* config, double* initdata
SimValueMatrix(const char* txt, const SimData& config
SimValueMatrix(const char* txt, const SimData* config
SimValueMatrix(const char* txt, const SimData& config, double* initdata
SimValueMatrix(const char* txt, const SimData* config, double* initdata
```

The constructor must be given a base configuration to copy to the new instance, in the form of a **SimData** instance. Each new **SimValueMatrix** also needs to have specified the initial number of rows (`height`) and columns (`width`), and whether it stores an array of column vectors (`asColumns=true`) or row vectors (`asColumns=false`).

An optional parameter is to specify whether a vector is "growable". If `setgrowable` is true (the default), then the matrix may be resized to any required number of cells. If it is set to false, then the `setmaxrows` and `setmaxcols` parameters specify the maximum number of rows and columns that are permitted in this matrix.

Initial values may be provided, by passing a reference to an array of type `double` to the `initdata` parameter. Care should be taken that this array is at least as long as the number of initial cells specified. The expected format is the the initial values are grouped in columns, as is the format provided by the MATLAB mex interface.

Labels may be assigned via the `txt` parameter.

8.3.2 Operators

As with **SimValue** , **SimValueMatrix** implements the basic mathematical operators. There are two types of operators, being the ones that perform an operation between two different operands and assign to a potentially different destination (such as $a = b + c$), and the ones that use the destination as one of the operands (such as $a+ = b$).

The $a = b+c$ type operators pass their result by value, rather then by reference. In the case of **SimValueMatrix** objects, this means creating a temporary variable to hold the result, and then the compiler creates a copy of that result before calling an assign operator to apply it to the destination variable. This is rather inefficient for code speed, but allows for simple programming especially in compound expressions.

```
SimValueMatrix operator +(const SimValueMatrix &a);
SimValueMatrix operator +(const SimValueMatrix *a);
SimValueMatrix operator +(const SimValue &a);
SimValueMatrix operator +(const SimValue *a);
SimValueMatrix operator +(double a);
SimValueMatrix operator +(long a);
SimValueMatrix operator +(int a);

SimValueMatrix operator -(const SimValueMatrix &a);
SimValueMatrix operator -(const SimValueMatrix *a);
SimValueMatrix operator -(const SimValue &a);
SimValueMatrix operator -(const SimValue *a);
SimValueMatrix operator -(double a);
SimValueMatrix operator -(long a);
SimValueMatrix operator -(int a);

SimValueMatrix operator *(const SimValueMatrix &a);
SimValueMatrix operator *(const SimValueMatrix *a);
SimValueMatrix operator *(const SimValue &a);
SimValueMatrix operator *(const SimValue *a);
SimValueMatrix operator *(double a);
SimValueMatrix operator *(long a);
SimValueMatrix operator *(int a);

SimValueMatrix operator /(const SimValue &a);
SimValueMatrix operator /(const SimValue *a);
SimValueMatrix operator /(double a);
SimValueMatrix operator /(long a);
SimValueMatrix operator /(int a);
```

NOTE : When another **SimValueMatrix** is supplied to the multiplication operator, it works as a MATLAB "dot multiply" function, multiplying corresponding elements of each element. To obtain a true matrix

multiplication, consider the `setWithMultiply` methods.

The equality type operators are called on the destination operand, which is also the first operand in the expression. Since these operators pass by reference, they are just as efficient as using standard methods.

```
SimValueMatrix &operator*=(const SimValueMatrix &a);
SimValueMatrix &operator*=(const SimValueMatrix *a);
SimValueMatrix &operator*=(const SimValue &a);
SimValueMatrix &operator*=(const SimValue *a);
SimValueMatrix &operator*=(double a);
SimValueMatrix &operator*=(long a);
SimValueMatrix &operator*=(int a);

SimValueMatrix &operator/=(const SimValueMatrix &a);
SimValueMatrix &operator/=(const SimValueMatrix *a);
SimValueMatrix &operator/=(const SimValue &a);
SimValueMatrix &operator/=(const SimValue *a);
SimValueMatrix &operator/=(double a);
SimValueMatrix &operator/=(long a);
SimValueMatrix &operator/=(int a);

SimValueMatrix &operator+=(const SimValueMatrix &a);
SimValueMatrix &operator+=(const SimValueMatrix *a);
SimValueMatrix &operator+=(const SimValue &a);
SimValueMatrix &operator+=(const SimValue *a);
SimValueMatrix &operator+=(double a);
SimValueMatrix &operator+=(long a);
SimValueMatrix &operator+=(int a);

SimValueMatrix &operator-=(const SimValueMatrix &a);
SimValueMatrix &operator-=(const SimValueMatrix *a);
SimValueMatrix &operator-=(const SimValue &a);
SimValueMatrix &operator-=(const SimValue *a);
SimValueMatrix &operator-=(double a);
SimValueMatrix &operator-=(long a);
SimValueMatrix &operator-=(int a);

SimValueMatrix &operator=(const SimValueMatrix &a);
SimValueMatrix &operator=(const SimValueMatrix *a);
```

NOTE : The following operators will resize the matrix to 1×1, and any old copies of the data reference will become invalid

```
SimValueMatrix &operator=(const SimValue &a);
SimValueMatrix &operator=(const SimValue *a);
SimValueMatrix &operator=(double a);
SimValueMatrix &operator=(long a);
SimValueMatrix &operator=(int a);
```

8.3.3 Direct Call Calculation Methods

These methods provide the same functionality as the overloaded operators, but may provide greater efficiency of compiled code, and are more flexible to use. Each method sets the called instance with the result of operating on the supplied parameters. The first parameter represents the left operand in an expresion, and the second parameter is the right operand. The "DotMultiply" routines are equivalent to the MATLAB `.*` statement, where corresponding pairs of elements are multiplied instead of performing a full vector multiplication.

```
void setWithDivide(const SimValueMatrix *a, const SimValue *b);
void setWithDivide(const SimValueMatrix *a, const SimValue &b);
```

```
void setWithDivide(const SimValueMatrix *a, long b);
void setWithDivide(const SimValueMatrix *a, int b);
void setWithDivide(const SimValueMatrix *a, double b);

void setWithAdd(const SimValueMatrix *a, const SimValueMatrix *b);
void setWithAdd(const SimValueMatrix *a, const SimValueMatrix &b);
void setWithAdd(const SimValueMatrix *a, const SimValue *b);
void setWithAdd(const SimValueMatrix *a, const SimValue &b);
void setWithAdd(const SimValueMatrix *a, long b);
void setWithAdd(const SimValueMatrix *a, int b);
void setWithAdd(const SimValueMatrix *a, double b);

void setWithSubtract(const SimValueMatrix *a, const SimValueMatrix *b);
void setWithSubtract(const SimValueMatrix *a, const SimValueMatrix &b);
void setWithSubtract(const SimValueMatrix *a, const SimValue *b);
void setWithSubtract(const SimValueMatrix *a, const SimValue &b);
void setWithSubtract(const SimValueMatrix *a, long b);
void setWithSubtract(const SimValueMatrix *a, int b);
void setWithSubtract(const SimValueMatrix *a, double b);

void setWithDotMultiply(const SimValueMatrix *a, const SimValueMatrix *b);
void setWithDotMultiply(const SimValueMatrix *a, const SimValueMatrix &b);
void setWithDotMultiply(const SimValueMatrix &a, const SimValueMatrix &b);
void setWithDotMultiply(const SimValueMatrix &a, const SimValueMatrix *b);

void setWithMultiply(const SimValueMatrix *a, const SimValue* b);
void setWithMultiply(const SimValueMatrix *a, const SimValue& b);
void setWithMultiply(const SimValueMatrix *a, int b);
void setWithMultiply(const SimValueMatrix *a, long b);
void setWithMultiply(const SimValueMatrix *a, double b);

void setWithMultiply(const SimValue *a, const SimValueMatrix*b);
void setWithMultiply(const SimValue &a, const SimValueMatrix*b);
void setWithMultiply(int a, const SimValueMatrix*b);
void setWithMultiply(long a, const SimValueMatrix*b);
void setWithMultiply(double a, const SimValueMatrix*b);

void setWithMultiply(const SimValueMatrix &a, const SimValue* b);
void setWithMultiply(const SimValueMatrix &a, const SimValue& b);
void setWithMultiply(const SimValueMatrix &a, int b);
void setWithMultiply(const SimValueMatrix &a, long b);
void setWithMultiply(const SimValueMatrix &a, double b);

void setWithMultiply(const SimValue *a, const SimValueMatrix&b);
void setWithMultiply(const SimValue &a, const SimValueMatrix&b);
void setWithMultiply(int a, const SimValueMatrix&b); void setWithMultiply(long a, const SimValueMatrix&b);
void setWithMultiply(double a, const SimValueMatrix&b);
```

8.3.4 Matrix Multiplication

The multiplications set the current matrix with the result, resizing it if allowed. If it is known and guaranteed that a particular quantity is a vector, then is should be passed in as a SimValueVector instead of a SimValue-Matrix.

The end of each of these declarations is
`, bool doNeg, bool trans1, bool trans2, bool item1left, bool allowResize);`, which has been omitted for clarity.

```
void setWithMultiply(const SimValueMatrix& mat1, const SimValueMatrix& mat2
void setWithMultiply(const SimValueVector& mat1, const SimValueMatrix& mat2
void setWithMultiply(const SimValueMatrix& mat1, const SimValueVector& mat2
void setWithMultiply(const SimValueVector& mat1, const SimValueVector& mat2
```

The following multiplications will create a new matrix and return it with a pointer. To avoid memory leaks, the object must be deleted when no longer needed. These call setWithMultiply, so the same comments apply as above.

The end of each of these declarations is
`, bool doNeg, bool trans1, bool trans2, bool item1multiplier);`, which has been omitted for clarity.

```
SimValueMatrix* doMult(const SimValueMatrix* mat1, const SimValueMatrix* mat2
SimValueMatrix* doMult(const SimValueMatrix* mat1, const SimValueVector* mat2
SimValueMatrix* doMult(const SimValueVector* mat1, const SimValueMatrix* mat2
SimValueMatrix* doMult(const SimValueVector* mat1, const SimValueVector* mat2
```

The following provide additional multiplication options. In the "MultBy" functions, the current instance is the left term, and in the "MultWith" functions the current instance is the right operand.

```
SimValueMatrix* transMultBy(const SimValueMatrix* mat2); // ie.  (this' * x)
SimValueMatrix* transMultBy(const SimValueVector* mat2); // ie.  (this' * x)
SimValueMatrix* transMultWith(const SimValueMatrix* mat1); // ie.  (x * this')
SimValueMatrix* transMultWith(const SimValueVector* mat1); // ie.  (x * this')

SimValueMatrix* negTransMultBy(const SimValueMatrix* mat2); // ie.  (this' * x)
SimValueMatrix* negTransMultBy(const SimValueVector* mat2); // ie.  (this' * x)
SimValueMatrix* negTransMultWith(const SimValueMatrix* mat1); // ie.  (x * this')
SimValueMatrix* negTransMultWith(const SimValueVector* mat1); // ie.  (x * this')

SimValueMatrix* negMultBy(const SimValueMatrix* mat2); // ie.  -(this * x)
SimValueMatrix* negMultBy(const SimValueVector* mat2); // ie.  -(this * x)
SimValueMatrix* negMultWith(const SimValueMatrix* mat1); // ie.  -(x * this)
SimValueMatrix* negMultWith(const SimValueVector* mat1); // ie.  -(x * this)
```

The next set of multiplications are for where the programmer can guarantee the output to be a vector. That is when `mat2` is a vector and `mat1` has the same number of columns as the length of `mat2`, or when `mat1` is a vector and `mat2` has the same number of rows as the length of `mat1`.

Each of the following are declared as `static`, and the end of each of these declarations is
`, bool doNeg, bool trans1, bool trans2, bool item1multiplier);`, which has been omitted for clarity.

```
static SimValueVector* doMultToGetVector(const SimValueMatrix& mat1, const SimValueVector& mat2
static SimValueVector* doMultToGetVector(const SimValueMatrix* mat1, const SimValueVector& mat2
static SimValueVector* doMultToGetVector(const SimValueMatrix& mat1, const SimValueVector* mat2
static SimValueVector* doMultToGetVector(const SimValueMatrix* mat1, const SimValueVector* mat2

static SimValueVector* doMultToGetVector(const SimValueVector& mat1, const SimValueMatrix& mat2
static SimValueVector* doMultToGetVector(const SimValueVector* mat1, const SimValueMatrix& mat2
static SimValueVector* doMultToGetVector(const SimValueVector& mat1, const SimValueMatrix* mat2
static SimValueVector* doMultToGetVector(const SimValueVector* mat1, const SimValueMatrix* mat2
```

8.3.5 Debugging Support

The **SimValueVector** class implements the **DebugInterface** and provides debugging support with the following :

- `void logItem(const char* desc=NULL);` - calls logItem of the **DebugSystem** instance, if defined

- `void dump();` - writes the value of the current instance to the main logs
- `void getCalcLabel(char* txt) const;` - returns the label of this instance
- `void getCalcLogValue(char* txt) const;` - returns a string for the calculation tracking and item logs, representing the value of the current instance.

8.3.6 Data Accessor Functions

- `void fillArray(double* d);` - fills the array referenced by `d` with doubles representing the values of the matrix. Values are grouped in columns, to be compatible with the format used by the MATLAB mex interface. Care must be taken to ensure the the array is large enough.
- `SimValue* getCell(int row, int col);` - Returns a reference to the cell indexed by `row` and `col`. Indexing is zero or one based, depending on the setting of the `indexFromOne` variable inherited from **DData**
- `SimValue* getCellIndex0(int row, int col) const;` - Returns a reference to the cell indexed by `row` and `col`. Indexing is zero based.
- `SimValueVector* getVector(int roworcol);` - Returns a reference to the vector indexed by `roworcol`. Indexing is zero or one based, depending on the setting of the `indexFromOne` variable inherited from **DData**
- `SimValueVector** getData();` - Returns a reference to the array of vectors
- `bool inCols() const;` - returns true if it the matrix consists of an array of column vectors.
- `int numRows() const;` - returns the current number of rows in the matrix.
- `int numCols() const;` - returns the current number of columns in the matrix.
- `SimValueVector* operator[](int i) const;` - same as `getVector(int roworcol);`
  NOTE : Be careful using the indexing operator. If the variable is a pointer to the **SimValueVector** then make sure that the code is accessing the operator, not indexing an array pointed to by the variable. e.g. if you declared `SimValueVector* f;` then you would need to use `(*f)[1].dump();` or `f[0][1].dump();`.
- `SimValue valueof();` - returns the value of the first cell in the first vector.

8.3.7 Vector Manipulation

The following methods will resize and fill the `dest` vector with the selected `rowOrCol`. Indexing is zero or one based, depending on the setting of the `indexFromOne` variable inherited from **DData** .

- `void setVectorWithRowOrCol(SimValueVector &dest,bool withCol, int rowOrCol, bool transposedMatrix=false, bool allowVectorResize=false, bool allowOrientationChange=false) const;`
- `void setVectorWithRowOrCol(SimValueVector *dest,bool withCol, int rowOrCol, bool transposedMatrix=false, bool allowVectorResize=false, bool allowOrientationChange=false) const;`

The optional `allowVectorResize` and `allowOrientationChange` specify whether the detination vector can be resized or reorientated, or whether it is already expected to be of the correct dimensions. The `withCol` option selects whether a row (false) or column (true) is selected, and if `transposedMatrix` is true then that row or column is taken from a transpose of the current matrix.

8.3.8 Matrix Manipulation

- `void resize(int newrows, int newcols);` - resizes the matrix, adding or deleting vectors and cells as needed
- `void setOrientation(bool isColumn);` - changes whether the matrix is a row vector or a column vector. Matrix is transposed if this changes.
- `void transpose();` - changes whether the matrix is a row vector or a column vector
- `void setVecMult(int col, int row, const SimValueVector& rowvector, const SimValueVector& colvector, bool item1multiplier, bool oneIsTransposed, bool doNeg);` - Sets the designated cell with the result of a vector multiplication between `rowvector` and `colvector`. `item1multiplier` specifies whether or not the first parameter is used as the multiplier term.

8.3.9 Miscellaneous Value assignment

- `void ones();` - sets all of the cells to 1
- `void zeros();` - sets all of the cells to 0
- `void generateRandom();` - sets all of the cells to random values
- `void generateIdentity();` - creates an identity matrix

9 Debugging Features

The C4Hardware library includes a number of debugging features that allow dumping of data to the screen, or one of several logs. These include

- Logging of data items in a format that can be read into Matlab as a ".m" file
- Ability to Log detail of every calculation that is performed
- Logging of data items in a custom format that allows for comparision with a Matlab version of the same algorithm

These are detailing in the following sections

9.1 Labels

Labels form one of the most useful parts of the debugging system, as they allow each object that extends **DData** to be given a descriptive text label. The **SimValue**, **SimValue**, and **SimValueMatrix** classes all provide the option to specify a label in their constructors. Any user created objects that extend **SimNumericType** should also provide this option.

9.1.1 Functionality

Most of the program code needed for handling labels is provided in the **DData** base class.

- `virtual void setLabel(const char* txt);` : Creates a local copy of the buffer refrenced by `txt`.
- `virtual void afterSetLabel();` : The version defined in **DData** does nothing. It is provided to allow objects to perform an action after a label change occurs. For example, to set the labels of any objects that it contains, such as cells within a vector.
- `void giveTempLabel();` : Creates a new label made up of the word "temporary" and the object's debugID
- `void setLabel(const char* txt, const char* txt2);` : Creates a local copy of the buffer refrenced by the concatenation of `txt` and `txt2`.
- `void copyLabel(char* txt);` : Copies the reference passed into the routine instead of copying the contents of the buffer. When using this, care should be taken to ensure that the label is not destroyed before the current instance is finished with it.
- `void copyLabelAsCopy(char* txt);` : calls `setLabel(txt," copy");`.
- `char* getID() const;` : Returns the reference to the label used by the current instance.
- `void copyLabelTo(char* txt) const;` : If a label exists, it is copied to the character buffer `txt`. If not, `txt` is filled with the words "Unlabelled id " with the debugID number.

These labels are then returned by objects implementing the **DebugInterface** class, for use by the **DebugSystem** .

9.1.2 Enabling labels

By default, the capacity to use labels is included in the C4Hardware classes, but is not enabled because it uses significant memory and slows performance. This is achieved via conditional define instructions given to the compiler preprocessor. The default defines are found in **c4hdefines.h** :

```
 #ifndef C4H_USELABELS
#define C4H_USELABELS 1;
#endif

#ifndef C4H_OBEYLABELS
#define C4H_OBEYLABELS 0;
#endif
```

The C4H_OBEYLABELS directive is the code that disables or enables the use of labels in the program code. This works by using conditional compiles around the relevant sections of code, so the relevant routines can be called, but will not do anything.

The C4H_USELABELS directive attempts to completely remove all reference to the labels from the compile code. If not used carefully, this may cause compilation problems because certain routines will no longer exist.

9.2 DebugInterface

DebugInterface is an abstract class that is implemented by **SimNumericType** , **SimValue** , **SimValueVector** , and **SimValueMatrix** . This means that the interface must be implemented by any class that extends **SimNumericType** . It defines some virtual methods that are used to obtain debugging information from these classes, without having to know exactly what type of class is being used.

```
 virtual void getCalcLabel(char* txt) const=0;
virtual void getCalcLogValue(char* txt) const=0;
```

The getCalcLabel method fills the character array in txt with a copy of the current object's descriptive label. This can generally be achieved by calling the copyLabelTo method that is inherited from **DData** :

```
void SimRealFloat::getCalcLabel(char* txt) const
{
copyLabelTo(txt);
}
```

The getCalcLogValue method fills txt with a human-readable string that provides details of the value of the quantity stored in the current class instance. For example, **SimRealFloat** implements this with :

```
sprintf(txt,"[Components = %1.20f, double=%1.20f, single=%1.20f,
hexdbl=%s]",value(),doubleprecision,singleprecision,hx);
```

The size of the buffer provided in both cases can be assumed to be 200 characters.

9.3 DebugSystem

The **DebugSystem** class provides several logging and tracking options to assist with the debugging and evaluation of C4Hardware projects. In typical use, a single **DebugSystem** instance is initialised, and passed to all **DData** based objects in the program. These objects can then make calls to the **DebugSystem** instance, which will then process these requests based on its configuration at initialisation.

There are currently five types of logging options available:

- Logging of data items to the screen or a file
- Logging of progress points to a file or screen

- Tracking of calculations performed during a program
- Alternative method of logging data items to a file
- Tracking of objects being created and destroyed, to void memory leaks

9.3.1 Constructor

There are two constructors available - one that provides all of the available options, and one that uses the defaults.

```
DebugSystem(bool trackreg, int fid, int dumpmode);
DebugSystem();
```

The parameters are as follows:

- `trackreg` : Set to true to track the registration and deregistration of objects (see **??**)
- `fid` : file id number - this is a number that is used in the filenames of the various data logs
- `dumpmode` : a set of flags that controls the logging mode to be used
  - bit 0 - 1=log to screen
  - bit 1 - 1=log data, progress points, and registration to a file
  - bit 2 - 1=create a file that tracks calculations throughout the program
  - bit 4 - 1=enable alternative item log

The default constructor logs to screen with no file logging and no registration tracking.

9.3.2 Registration Tracking

```
long registerItem(int idtype);
void deregisterItem(long idnum);
```

The registration tracking option logs the creation and deletion of **`DData`** based objects in the main data log. When registered, each object is given a unique debugID that is used to make sure the object is later deleted, and only deleted once. Creation is detected by adding a call to `registerItem` in the constructors of the relevant objects, and deletion is monitored via a call to `deregisterItem` with the assigned debugID.

If an item is deleted more than once, or any items have not been deregistered when the **`DebugSystem`** destructor is called, an error is added to the main log.

9.3.3 Main Data Log

The main data log is the most flexible logging option, and allows logging of item values, status strings, and other descriptive text. It is enabled in the constructor by setting the appropriate `dumpmode` bits and providing a `fid` value.

If the `fid` provided to the constructor is positive, then the file name will be **`C:\c4test####.txt`** on Windows systems or **`~/c4test####.txt`** on Linux systems, where #### is replace by `fid`. If `fid` is zero or less, then the filenames will be **`C:\c4example.txt`** or **`~/c4example.txt`** respectively.

There are a number of methods that can be called to write to this log :

```
void dostat(const char* g);
void dostat(const char* g, int i);
void dostat(const char* g, double i);
void dostat(const char* g, int i, int i2);
void dostat(const char* g, long i, long i2);
void dostat(const char* g, long i);
```

The dostat routines are called with an printf expression in g, and up to two numerical parameters.

```
void dumpfloat(const char* g, double f);
```

This dumpfloat method is the same as the corresponding dostat method, and is kept as a legacy item.

Additionally, the void useScreen(bool useit); method can be used to turn screen logging on an off after initialisation.

9.3.4 Progress Points

Progress points are logged to all of the log files that are open at the time of being called. A special routine is provided for the tracking of progress points throughout a program.

```
void progressPoint(int callerID, const char* txt, int item1, int item2, int item3, int item4,
int item5);
```

calledID should be the debugID of the calling object, and txt is a plain string of readable text. The five items will be logged as a comma seperated list if they are non-negative.

Progress point lines are preceeded by a % sign, so that they appear as comments if the log file is used as a MATLAB ".m" file.

9.3.5 Alternative Item Log

The Item Log is designed to log values in a standard format that is easily parsed by another script or program that might need to analyse the results. It is enabled in the constructor by setting the appropriate dumpmode bits and providing a fid value.

If the fid provided to the constructor is positive, then the file name will be **C:\itemlog####.txt** on Windows systems or **~/itemlog####.txt** on Linux systems, where #### is replace by fid. If fid is zero or less, then the filenames will be **C:\itemlog.txt** or **~/itemlog.txt** respectively.

The following methods are available for writing to the Item Log : `void logitem(const DebugInterface* dest,const char* desc,const char* indent="");`

```
void logitem(long dest,const char* desc,const char* indent="");
void logitem(int dest,const char* desc,const char* indent="");
void logitem(double dest,const char* desc,const char* indent="");
```

The parameters are as follows :

- dest : the source of the value to be logged. If a **DebugInterface** is supplied, this also logs the label associated with that item.
- desc : A text string providing an additional description or context for the item (e.g. "Initial Value")
- indent : Specifies an indent before the log line is printed. Used for logging subitems of a container object - such as cells of a vector.

Log strings are formatted as either

- <indent>Item <##> - <desc> : <label> = <value>
- <indent>Item <##> - <desc> : <value>

In many cases, this routine is note called directly, but instead a void logItem(const char* desc=NULL); method is provided in the **SimValueMatrix** , **SimValueVector** , and **SimValue** classes.

9.3.6 Tracking of Calculations

The calculation tracker provides detailed information about almost every calculation that occurs in **DData** based classes. It is enabled in the constructor by setting the appropriate dumpmode bits and providing a fid value.

If the fid provided to the constructor is positive, then the file name will be **C:\calclog####.txt** on Windows systems or **~/calclog####.txt** on Linux systems, where #### is replace by fid. If fid is zero or less, then the filenames will be **C:\calclog.txt** or **~/calclog.txt** respectively.

Log entries are created automatically when operation routines are called on **SimValueMatrix**, **SimValueVector**, and **SimValue** objects. There are two groups of methods to support this feature, with the first being intended for assignment operations :

```
void assign(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt, bool showvalues=true);
void assign(const DebugInterface* dest,const char* assgn,long srclt, bool showvalues=true);
void assign(const DebugInterface* dest,const char* assgn,double srclt, bool showvalues=true);
void assign(const DebugInterface* dest,const char* assgn,int srclt, bool showvalues=true);
```

The parameters to these methods include

- dest : is the destination item that will hold the final result
- assgn : is a text string that represent the assignment operation being performed (e.g. " = ")
- srclt : the quantity that is to be assigned to dest
- showvalues : If true, a summary of the initial values of each quantity will be logged.

The second group of methods split the logging into two parts. Before the calculation is performed, a call to a startcalc routine is made to log the initial state of the variables. Once the calculation is complete, a call to endcalc is made to log the completion, optionally logging the final value.

```
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt, bool
showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,long srclt, bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,double srclt, bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,int srclt, bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn, bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt,const
char* op,const DebugInterface* srcmd,const char* op2,const DebugInterface* srcrt,bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt,const
char* op,const DebugInterface* srcrt,bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt,const
char* op,const int srcrt,bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt,const
char* op,const long srcrt,bool showvalues=true);
void startcalc(const DebugInterface* dest,const char* assgn,const DebugInterface* srclt,const
char* op,const double srcrt,bool showvalues=true);
void startcalc(const char* assgn);
void endcalc(const DebugInterface* dest);
void endcalc();
```

An advantage of using these paired methods is that the start and end methods trigger nesting of calculations within the log file. A startcalc call will increase nesting indenting, and endcalc removes it. This allows, for example, individual calculations within a vector multiplication to be nested within a log entry for that overall calculation.

Additional parameters include

- srclt : the left operand in an operation

- `op, op2` : text strings representing the operation in progress (e.g. +, *, /)
- `srcmd` : the middle operand, where used
- `srcrt` : the right operand

10 Exception Throwing

C4Hardware provides a mechanism for generating descriptive error conditions (exceptions) and providing a traceback of the call stack at the point that the error was triggered. This is done with the **ExceptionSystem** class.

The exceptions are enabled or disabled via a set of compiler `#DEFINE`s, which have default values assigned in **c4hdefines.h**.

```
#ifndef C4H_USE_EXCEPTIONSYSTEM
#define C4H_USE_EXCEPTIONSYSTEM 1
#endif

#ifndef C4H_SIMVALUE_EXCEPTION_TRACEBACKS
#define C4H_SIMVALUE_EXCEPTION_TRACEBACKS 1
#endif

#ifndef C4H_SIMVALUEVECTOR_EXCEPTION_TRACEBACKS
#define C4H_SIMVALUEVECTOR_EXCEPTION_TRACEBACKS 1
#endif

#ifndef C4H_SIMVALUEMATRIX_EXCEPTION_TRACEBACKS
#define C4H_SIMVALUEMATRIX_EXCEPTION_TRACEBACKS 1
#endif
```

To override these settings, the user should assign new values by passing appropriate parameters to the C++ compiler. For example, in gcc, use the `-D` switch:

```
gcc -DC4H_USE_EXCEPTIONSYSTEM=0 file1 file2 file3...
```

The `C4H_USE_EXCEPTIONSYSTEM` directive enables or disables the entire system. If set to 1, it is enabled and **ExceptionSystem** objects can be thrown and caught. Otherwise, any exceptions should be thrown as plain `int` types. The `TRACEBACK` directives indicate whether or not the respective class types should attempt to catch and rethrow exceptions to create the traceback described below.

10.1 Using Exceptions

Exceptions are thrown by instantiating a new instance of the **ExceptionSystem** class and throwing its reference. The basic constructor allows the specification of a code number, the class name, and a description that would typically identify the execution point.

```
throw new ExceptionSystem(ExceptionSystem::EX_IncorrectNumericType,"SimRealFloat","doDivide");
```

IMPORTANT : ExceptionSystem objects must be thrown as references, otherwise they will not be caught in the traceback system. It is also more efficient to throw references, rather than copying objects.

ExceptionSystem objects are typically thrown in the **SimValue**, **SimValue**, and **SimValueMatrix** classes, as well as in custom numeric types that extend **SimNumericType** . It is also possible for **DualValueCalculator** and **SingleValueCalculator** classes to throw an **ExceptionSystem**.

The **SimValue**, **SimValue**, and **SimValueMatrix** classes are designed to catch **ExceptionSystem** objects thrown from other objects that they call. When they catch an exception, they call the `addExceptionPoint` method to add a step to the traceback, and then rethrow the exception.

```
catch (ExceptionSystem* e)
{
      e->addExceptionPoint("SimValueVector","vectorMultiply");
      throw e;
}
```

The rethrown exception is then possibly caught by another point further up the call stack, which adds another exception point, rethrows the excpetion, and so on. At the top level, the use program should use a `try` / `catch` construct to catch any exceptions that might be thown. Once caught, exceptions should be handled and then destroyed.

```
try {
      //user code
}catch (ExceptionSystem* e) {
      e->dumpData();
      delete e;
}
```

IMPORTANT : Failing to `delete` the exception will cause a memory leak.

The result of the `dumpData` call will be show details of the exception that occurred, and the points in the code that added to the traceback list

```
Exception EX_DivideByZero
      SimRealFloat, doDivide(const SimRealFloat& a) :
            (Unlabelled id 10) :  [Components = 0.00000000000000000000]
      SimValue, doDivide(const SimNumericType& a)
            (Unlabelled id 10) :  [Components = 0.00000000000000000000]
      SimValue, operator /(SimValue&)
            (Unlabelled id 10) :  [Components = 0.00000000000000000000]
```

10.2 ExceptionSystem

The **`ExceptionSystem`** class is the only exception related class that is directly used by other parts of the software. It is the object that is thrown by reference when an exception is triggered, and controls the traceback list and details of the exception. Most of this is done via an array of **`ExceptionPoint`** objects.

10.2.1 Static Constants

These constants reprents the currently defined exceptions :

- `EX_IncorrectNumericType` - An object was passed as a **`SimNumericType`** , but was the incorrect derivative of that class.
- `EX_SquareRootOfNegative` - attempt to calculate $\sqrt{x}$, with $x < 0$ unsupported by numeric type
- `EX_InverseOfZero` - attempt to calculate $\frac{1}{0}$ unsupported by numeric type
- `EX_MantissaTooLarge` - signifies an internal error in the **`SimRealFloat`** numeric class
- `EX_MaximumSizeExceeded` - when a vector/matrix is resized above its predefined maximum
- `EX_InvalidIndexSelected` - attempt to index a cell not contained in a matrix or vector
- `EX_VectorDimensionsMismatch` - Dimensions of vector(s) specified unsitable for called operation
- `EX_MatrixDimensionsMismatch` - Dimensions of matrix specified unsitable for called operation
- `EX_OrientationDoesNotMatchExistingConfiguration` - An attempt to resize a matrix or vector was called, but specified a different row/column orientation than what is already used.
- `EX_SquareMatrixExpected` - Operation requires a square matrix
- `EX_ColumnVectorExpected` - Operation requires a column vector
- `EX_RowVectorExpected` - Operation requires a row vector

- `EX_MatrixMustBeOneRowSmallerThanVector` - The Matrix provided needs to have one less row than is used in the specified vector.
- `EX_VectorIsTooSmall` - The provided vector is not large enough for the requested operation
- `EX_DivideByZero` - attempt to calculate $\frac{x}{0}$ unsupported by numeric type

10.2.2 Constructors

A variety of constructors are available, depending on how much information is to be provided in the error messages.

- `ExceptionSystem(int exceptionID, const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, const DebugInterface* item3, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `ExceptionSystem(int exceptionID, const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `ExceptionSystem(int exceptionID, const char* cname, const char* pid, const DebugInterface* item1, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `ExceptionSystem(int exceptionID, const char* cname, const char* pid, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`

Each of the constructors vary only in the number of `const DebugInterface*` parameters that are passed.

- `exceptionID` - a code number that identifies which type of error occurred. This should be set to one of the `EX_` constants.
- `cname` - a null terminated text string containing the name of the class throwing the excpetion
- `pid` - a null terminated text string that identifies the point in the class where the exception occurred. Typcially this will include the method name.
- `item1`, `item2`, `item3` - If these are provided, the current state of these objects will be reported by the **`ExceptionSystem`**
- `items` - How many numerical items to report (below)
- `d1`, `d2`, `d3`, `d4`, `d5` - if the setting of `items` allows, these will be reported in a comma seperated list after the exception description.

The following copy constructors are also available, but should not be needed

```
 ExceptionSystem(const ExceptionSystem *src);
ExceptionSystem(const ExceptionSystem &src);
```

10.2.3 Public Methods

The following methods are used to add steps to the traceback list.

- `void addExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, const DebugInterface* item3, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `void addExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`

- `void addExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `void addExceptionPoint(const char* cname, const char* pid, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`

These are used within **SimValue**, **SimValue**, and **SimValueMatrix** when an exception is caught and rethrown. They may also be used within the user program as required. All parameters have the same meanings as for the constructors.

The next method is used for reporting information about the exception :

```
void dumpData(FILE *fid=NULL);
```

If a file pointer is provided, that is used, otherwise the output is to the standard output. This method prints the exception id, along with any information provided in the constructor and `addExceptionPoint` calls. It also prints the traceback list to show where in the program the offending piece of code was called.

10.3 ExceptionPoint

The **ExceptionPoint** class is simply a data container for steps in the traceback list maintained by **ExceptionSystem** .

10.3.1 Constructors

The constructors are identical in parameters to the `addExceptionPoint` methods in **ExceptionSystem** .

- `void ExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, const DebugInterface* item3, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `void ExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, const DebugInterface* item2, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `void ExceptionPoint(const char* cname, const char* pid, const DebugInterface* item1, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`
- `void ExceptionPoint(const char* cname, const char* pid, int items=0, double d1=0, double d2=0, double d3=0, double d4=0, double d5=0);`

10.3.2 Public Methods

The following method is used for reporting information about the current step in the exception traceback :

```
void dumpData(FILE *fid=NULL);
```

If a file pointer is provided, that is used, otherwise the output is to the standard output.

11 Statistics Counter

The `StatCounter` class is currently unused, but it is intended to provide an abstract interface for the purpose of tracking statistics of an algorithm. For example, it might count number of multiplications or other operations, average number of iterations, etc.

12 Testbench Writer

The `TestbenchWriter` class is awesome, but I haven't documented it yet :-)

Coming soon...

Part III

Using C4Hardware

13 Getting Started

The C4Hardware provide a "middle layer" of classes, so it is up to the programmer to create the top and lower layers of classes to complete their project.

The lower layer of classes specify the specific behaviour of the number system that is to be modelled. While some example classes are provided with the C4Hardware library, it is likely that the programmer will want to customise these to meet their own requirements.

The upper layer of classes implement the actual algorithm, using the C4Hardware interface. Very rarely should the program need to directly access the lower layer classes, so that these classes may be substituted as seamlessly as possible.

13.1 The C4Hardware Automatic Configuration System

The easiest way to get started with C4Hardware is to create an instance of the **c4HardwareConfiguration** class, and use that to perform all of the configuration tasks. An example of its use is as follows :

```
 int sysid = -1;
c4HardwareConfiguration* c4hardware = new c4HardwareConfiguration();
c4HDLRealFloatHandler::registerNumberSystem(c4hardware);
sysid = c4hardware->createSystem("c4HDLRealFloat","mantissa=6;exponent=6");
```

The first line defines a variable that acts like a handle to a specific instance of a numeric configuration. That is, it is passed into various procedures to specify on which which numeric system to get/set values. This will be demonstrated shortly.

The second line simply creates a new instance of the **c4HardwareConfiguration** class. There are no parameters into the constructor. This class instance will handle all of the configuration information transparently to the user and programmer.

The third line is used to tell the **c4HardwareConfiguration** class about a particular number system. Initially, the **c4HardwareConfiguration** class knows nothing about any numeric system, so this static call is needed on each data type that is to be used.

Finally, an instance of a numeric system can be created by calling createSystem on the **c4HardwareConfiguration** class. The first parameter is a string that identifies the numeric type that you wish to use. The second argument is a semicolon separated list of parameters that are understood by that particular numeric type. The handle variable is then assigned a value to uniquely identify that particular configuration.

Additional numeric systems may be defined within the same project. Each will be given its own handle to uniquely identify it.

13.1.1 Setup a debugger

```
c4hardware->addDebugger(sysid,new DebugSystem(false,0,1), true);
```

This is an optional step. The debugger class is described later in this document, and allows an easy means to add various logging and value dumping features to your code.

The addDebugger command takes three arguments. The first is the handle to the individual numeric system. The second is a reference to a **DebugSystem** class. This may either be a predefined instance or, as in this example, declared as a new item inline. The third argument determines whether or not to delete the instance from memory when the numeric system is destroyed. Setting this value to true allows the memory management to be handled internally by the predefined classes, without additional interation by the programmer.

13.1.2 Setup a Statistics Module

```
c4hardware->addStats(sysid,new StatCounter(false,1,12,13,14), true);
```

This is an optional step. The statistics class is described later in this document, and allows an easy means to add tracking of data statistics to your code.

The addStats command takes three arguments. The first is the handle to the individual numeric system. The second is a reference to a **StatCounter** class. This may either be a predefined instance or, as in this example, declared as a new item inline. The third argument determines whether or not to delete the instance from memory when the numeric system is destroyed. Setting this value to true allows the memory management to be handled internally by the predefined classes, without additional interation by the programmer.

13.1.3 Setup a Testbench Writer

```
c4hardware->addTestbench(sysid,new TestbenchWriter(true, doAppend, dumpmode, "d:\testdata\",
"testb", probid, ".txt"), true);
```

This is an optional step. The **TestbenchWriter** class is described later in this document, and allows the logging of testvectors for input or checking with a corresponding HDL model.

The addTestbenchs command takes three arguments. The first is the handle to the individual numeric system. The second is a reference to a **TestbenchWriter** class. This may either be a predefined instance or, as in this example, declared as a new item inline. The third argument determines whether or not to delete the instance from memory when the numeric system is destroyed. Setting this value to true allows the memory management to be handled internally by the predefined classes, without additional interation by the programmer.

13.2 Manually Initialising the C4Hardware Interface

To use the C4Hardware libraries via manual initialisation, a number of initialisation tasks need to be performed. An example initialisation code is given below, based on what the automatic system would do, and is described in the following subsections.

```
 DebugSystem* dbs;
dbs = new DebugSystem(false,probid,dumpmode);

// Configure the SimNumericConfiguration
SimRealFloatConfiguration numericconfig;

// Create an EmulationData
EmulationOverrides emulation;

// Setup and configure a Simdata
SimData config2;
config2.setEmulationData(&emulation);
config2.debugger = dbs;

// Create a SimNumericType
SimRealFloat examplenumber(&numericconfig,config2);
config2.setBaseConfig(&examplenumber);

numericconfig.configureNumberSystem(configbits,8,true);
config2.setIndexing(true);
```

13.2.1 Setup a debugger

```
 DebugSystem* dbs; dbs = new DebugSystem(false,probid,dumpmode);
```

This is an optional step. The debugger class allows an easy means to add various logging and value dumping features to your code.

13.2.2 Configure the SimNumericConfiguration

```
// Configure the SimNumericConfiguration
SimRealFloatConfiguration numericconfig;
```

The first item that needs to be set up is information that is specific to the numerical system that you want to use. In this example, and instance of **SimRealFloatConfiguration** is defined, which is based on the **SimNumericConfiguration** class. The specific **SimNumericType** that you use will depend on the choice of **SimNumericType** chosen, as each **SimNumericType** depends on a particular **SimNumericConfiuration**.

13.2.3 Create an EmulationData

```
// Create an EmulationData
EmulationOverrides emulation;
```

The **EmulationData** class is a general class that allows you to override certain numerical operations if you wish to perform them in a non-standard manner. For example, you may wish to override the square root operation with a suboptimal algorith that runs faster in hardware.

Even if no overrides are desired, this class is required to set the NULL defaults.

13.2.4 Setup and configure a SimData

```
// Setup and configure a SimData
SimData config2;
config2.setEmulationData(&emulation);
config2.debugger = dbs;
```

The **SimData** base class inherits **DData** provides pointers to various sources of configuration data, and is inherited by **SimValue**, **SimValueVector** and **SimValueMatrix**. In addition to the contents of **DData**, it provides a pointer to the base numerical type so that these three classes know what numerical class to use when creating new object instances.

Two items that must be set, and can be set at this point, are references to the EmulationData, and a reference to the debugging system. There are other items that need to be configured, but these cannot be done yet.

13.2.5 Create a SimNumericType

```
// Create a SimNumericType
SimRealFloat examplenumber(&numericconfig,config2);
config2.setBaseConfig(&examplenumber);
```

The **SimNumericType** represents a single numerical item, such as an integer, a floating point number, a complex number, etc. In this example, a **SimRealFloat** is created as an example of a type that inherits **SimNumericType**. Typically, each **SimNumericType** will require a reference to a corresponding instance of **SimNumericConfiguration** that is specific to the **SimNumericType** being used.

This particular instance is created as a "base configuration", that will allow creation of other objects of the same type. Since the C4Hardware classes do not know anything about the specific **SimNumericType** being used, they rely on a virtual `duplicate()` method in the **SimNumericType** instance to create new values, vectors, and matrices of the correct type.

Once this object is created, it is necessary to configure the **SimData** object with a reference to this base configuration, using the `setBaseConfig` method.

13.2.6 Other initialisation

```
 numericconfig.configureNumberSystem(configbits,8,true);
config2.setIndexing(true);
```

There will likely be other initialisation that you need to do, specific to your needs and the numerical type(s) being used. In this example, the floating point data type is configured with a particular number of mantissa bits, exponent bits, and whether it is signed. The SimData is configured to use MATLAB indexing mode, where the first row/column of a vector or matrix is numbered "1".

14 Building a Custom Numeric Type

To build a numeric data type that works with the C4Hardware interace, there are a number of guidelines to be followed. There are three main classes that you need to implement, each derived from an abstract base class :

- **SimNumericConfiguration** - store data about how the numeric type should behave in simulation.
- **SimNumericType** - implements the methods that instantiate the data type's behaviour, and contains information about individual instances of the numeric type (in, particular it's current value).
- **SimNumericTypeHandler** - This is optional, but its inclusion allows for ease of intergration into the automatic configuration system.

14.1 Creating a SimNumericConfiguration

The main reference on the **SimNumericConfiguration** class is in Section 5.1. An example is the **SimRealFloatConfiguration** class, which extends the base class as follows :

```
class SimRealFloatConfiguration :  public SimNumericConfiguration
```

The class should contain data about the configuration of the numeric type, which could be shared amongst several instances. This could include parameters such as the number of bits and how to handle overflows and other anomalies.

All instances must implement `void copyConfig(const SimNumericConfiguration& src)` such that it copies all configuration data from the `src` object to the current instance. Typically, this method will call `copyConfigBase(src)`, and then copy any other data that is specific to the derived class.

The constructor must set `configType` to a value that is unique to this derived class. You should add new constants to **SimNumericConfiguration** as required.

14.2 Creating a SimNumericType

The main reference on the **SimNumericType** class is in Section 7.An example is the **SimRealFloat** class, which extends the base class as follows :

```
class SimRealFloat :public SimNumericType, public SimRealFloatData
```

To work correctly with the C4Hardware libraries, it must inherit both the **SimNumericType** base class, and the **SimNumericTypeData** class that was created above. It must then implement each of the virtual **SimNumericType** methods, even if they aren't used or are irrelevant to the data type. For example, `imagValue()` needs to be implemented, even though it will always return zero for a real-only numeric type.

The **SimNumericType** class also inherits the abstract **DebugInterface** class, but does not implement any of its virtual functions. Therefore, these must also be implemented when creating a class based on **SimNumericType** . These are detailed in Section 9.2.

A **SimNumericType** should also implements the exception throwing mechanism (Section 10) provided by C4Hardware , unless it is intended that it never be used with this numeric type. At relevant points in the code, the compiler `define`s should be checked and the appropriate exception thrown :

```
#if C4H_USE_EXCEPTIONSYSTEM == 1
      throw new ExceptionSystem(ExceptionSystem::EX_IncorrectNumericType,"SimRealFloat","doDivide");
#else
      throw ExceptionSystem::EX_IncorrectNumericType;
#endif
```

The class should also contain data that is specific to individual instances of the numeric type - such as the value of its numerical components (mantissa, exponent, etc).

14.2.1 Constructor

There are several special tasks that the constructor must perform. The constructor from **SimRealFloat** will be used as an example.

```
SimRealFloat::SimRealFloat(SimRealFloatConfiguration* base, const DData& config,char* txt, bool labelAsCopy)
```

The declaration is up to the programmer, but will need certain parameters before the data type can be used. In the above example,

- `SimRealFloatConfiguration* base` provides the **SimNumericConfiguration** class to be used with this data type. This is needed to set the `numeric` variable, which must be set before the instance can be used.
- `const DData& config` specifies a source for the general emulation data that is not specific to the numeric type. This data needs to be copied by the constructor.
- `char* txt, bool labelAsCopy` provides a text description of the data type, used for indentifying it in data logs.

The first task of the constructor body is to configure the debugger :

```
if (config.debugger != NULL) {
      debugID = config.debugger->registerItem(DebugSystem::dbFLOATVALUE);
      if (debugID>0) {
            debugger = config.debugger;
      }
} else
      debugID = -1;
```

The `debugID` provides a unique identifier for this object, which can be used to track it in debugging logs. If the debugger exists in the supplied **DData** , then it should be used to register the object and obtain an ID. The `debugger` variable needs to be explicitly set by the constructor, as it will not be copied in the following line.

```
copyDDataConfig(config);
```

This line copies most of the data from the supplied **DData** to the **DData** variables inherited in the current instance.

```
if (labelAsCopy)
      copyLabel(txt);
else
      setLabel(txt);
```

An optional feature of the debugger is the ability to set descriptive labels for inividual items of data. These labels appear in logs along with the corresponding values. Labels can be set as a “copy”, where the reference is copied to the new object, or by creating a new string for this instance. If the “copy” mode is used, care must be taken not to destroy the original label before this instance is finished with it.

```
obtype = OB_FloatValue;
numericType = NT_REALFLOAT;
```

These two variables should be set in the classes extended from **SimNumericConfiguration** and **SimNumericTypeData** respectively, although it doesn’t hurt to properly set here.

```
numeric = base; myConfig = base;
```

These assignments set the `numeric` variable inherited from **SimNumericTypeData** , and also the variable `myConfig` that is stored in **SimRealFloat** . Storing two copies is a little wasteful, but `myConfig` is declared as a **SimRealFloatConfiguration** , and this improves code readability by avoiding having to cast the pointer every time that it needs to be used as a **SimRealFloatConfiguration** instead of a **SimNumericConfiguration**

14.3 Creating a SimNumericTypeHandler

Part IV
Predefined Numeric Types

15 SimRealFloat

The **SimRealFloat** class provides a base numeric type for a real valued floating point system, with restricted bit precision. It provides a working example of how the C4Hardware library functions.

SimRealFloat maintains three different representation of a quantity's value, and each are maintained regardless of the simulation mode. The main representation is a "components mode" system, where the mantissa, exponent, and sign bit of a floating point number are stored in separate variables and individually managed by the **SimRealFloat** class.

Alternatively, C++ double and single precision representations are also maintained. These are stored in `double` and `single` C++ data types, and updated using the built-in C++ operators and standard function calls. The purpose of this representation is to allow a sanity check against the custom model, and also to allow the triggering of break points or other alerts when the restricted precision model deviates too far from the double precision model.

The **SimRealFloat** class is still incomplete, and items that need to be added or refined include:

- Currently, exponents are unlimited in size, despite the configuration parameter
- All values are signed, regardless of the configuration parameter
- Need to define how to handle exponent overflows in emulate mode

15.1 SimRealFloatData

The **SimRealFloatData** class implements **SimNumericTypeData** for the **SimRealFloat** numeric type :

```
class SimRealFloatData :  public virtual SimNumericTypeData
```

All of the **SimRealFloatData** items are stored in protected variables

- `double mantissa;` - the mantissa of the components mode representation
- `long exponent;` - the exponent of the components mode representation
- `bool neg;` - the sign bit of the components mode representation
- `double doubleprecision;` - the C++ double precision representation
- `float singleprecision;` - the C++ single precision representation

The constructor takes no parameters and there is only one method, a copy routine :

```
void copyData(const SimRealFloatData &src);
```

15.2 SimRealFloatConfiguration

The **SimRealFloatConfiguration** class extends **SimNumericConfiguration** and defines the following configuration data :

- `int manbits;` - how many mantissa bits are allowed in hardware emulation mode
- `int expbits;` - how many exponent bits are allowed in hardware emulation mode
- `double manbitval;` - 2^{manbits}
- `bool dosigned;` - whether to allow signed numbers (as opposed to positive only)

There are also some additional variable that are not currently used.

15.2.1 Public Methods

There are two available public methods. One is a simply copy routine, and the other provides configuration data for the numeric type.

```
void copyConfig(const SimNumericConfiguration &orig);
void SimRealFloatConfiguration::configureNumberSystem(int mantissaBits, int exponentBits, bool
isSigned);
```

15.3 SimRealFloat

The **SimRealFloat** class implements **SimNumericType** and extends **SimRealFloatData** for this numeric type, and the documentation for those data types should be referenced for further information on the available methods.

```
class SimRealFloat :  public SimNumericType, public SimRealFloatData
```

Most of the numerical operations are performed by manuipulating the exponent and mantissa as required, and then calling the `resync()` method. This method is provides common code for adjusting the mantissa and exponent to make sure that the quantity is represented in a standard floating point form. (specifically, that the mantissa is at least 1 and less than 2).